
PyTorch-Lightning-Bolts Documentation

Release 0.3.0

PyTorchLightning et al.

Mar 29, 2021

START HERE

1	Installation	1
2	Introduction Guide	3
3	Model quality control	13
4	Build a Callback	17
5	Info Callbacks	19
6	Self-supervised Callbacks	23
7	Variational Callbacks	25
8	Vision Callbacks	27
9	DataModules	31
10	Sklearn Datamodule	33
11	Vision DataModules	37
12	Datasets	53
13	AsynchronousLoader	57
14	Losses	59
15	Object Detection	61
16	Reinforcement Learning	63
17	How to use models	65
18	Classic ML Models	73
19	Autoencoders	77
20	Convolutional Architectures	83
21	GANs	89
22	Reinforcement Learning	95

23 Self-supervised Learning	121
24 Learning Rate Schedulers	141
25 Self-supervised learning Transforms	143
26 Self-supervised learning	155
27 Semi-supervised learning	159
28 Self-supervised Learning Contrastive tasks	161
29 Contributing	165
30 PL Bolts Governance Persons of interest	169
31 Changelog	171
32 Indices and tables	177
Python Module Index	393
Index	397

INSTALLATION

You can install using `pip`

```
pip install pytorch-lightning-bolts
```

Install bleeding-edge (no guarantees)

```
pip install git+https://github.com/PytorchLightning/pytorch-lightning-bolts.  
↪git@master --upgrade
```

In case you want to have full experience you can install all optional packages at once

```
pip install pytorch-lightning-bolts["extra"]
```


INTRODUCTION GUIDE

Welcome to PyTorch Lightning Bolts!

Bolts is a Deep learning research and production toolbox of:

- SOTA pretrained models.
- Model components.
- Callbacks.
- Losses.
- Datasets.

The Main goal of Bolts is to enable trying new ideas as fast as possible!

All models are tested (daily), benchmarked, documented and work on CPUs, TPUs, GPUs and 16-bit precision.

some examples!

```
from pl_bolts.models import VAE, GPT2, ImageGPT, PixelCNN
from pl_bolts.models.self_supervised import AMDIM, CPCV2, SimCLR, MocoV2
from pl_bolts.models import LinearRegression, LogisticRegression
from pl_bolts.models.gans import GAN
from pl_bolts.callbacks import PrintTableMetricsCallback
from pl_bolts.datamodules import FashionMNISTDataModule, CIFAR10DataModule, 
↳ ImagenetDataModule
```

Bolts are built for rapid idea iteration - subclass, override and train!

```
from pl_bolts.models import ImageGPT
from pl_bolts.self_supervised import SimCLR

class VideoGPT(ImageGPT):

    def training_step(self, batch, batch_idx):
        x, y = batch
        x = _shape_input(x)

        logits = self.gpt(x)
        simclr_features = self.simclr(x)

        # -----
        # do something new with GPT logits + simclr_features
        # -----

        loss = self.criterion(logits.view(-1, logits.size(-1)), x.view(-1).long())
```

(continues on next page)

(continued from previous page)

```
logs = {"loss": loss}
return {"loss": loss, "log": logs}
```

Mix and match data, modules and components as you please!

```
model = GAN(datamodule=ImagenetDataModule(PATH))
model = GAN(datamodule=FashionMNISTDataModule(PATH))
model = ImageGPT(datamodule=FashionMNISTDataModule(PATH))
```

And train on any hardware accelerator

```
import pytorch_lightning as pl

model = ImageGPT(datamodule=FashionMNISTDataModule(PATH))

# cpus
pl.Trainer.fit(model)

# gpus
pl.Trainer(gpus=8).fit(model)

# tpus
pl.Trainer(tpu_cores=8).fit(model)
```

Or pass in any dataset of your choice

```
model = ImageGPT()
Trainer().fit(
    model,
    train_dataloader=DataLoader(...),
    val_dataloader=DataLoader(...)
)
```

2.1 Community Built

Then lightning community builds bolts and contributes them to Bolts. The lightning team guarantees that contributions are:

1. Rigorously tested (CPUs, GPUs, TPUs).
 2. Rigorously documented.
 3. Standardized via PyTorch Lightning.
 4. Optimized for speed.
 5. Checked for correctness.
-

2.1.1 How to contribute

We accept contributions directly to Bolts or via your own repository.

Note: We encourage you to have your own repository so we can link to it via our docs!

To contribute:

1. Submit a pull request to Bolts (we will help you finish it!).
2. We'll help you add [tests](#).
3. We'll help you refactor models to work on ([GPU](#), [TPU](#), [CPU](#))..
4. We'll help you remove bottlenecks in your model.
5. We'll help you write up [documentation](#).
6. We'll help you pretrain expensive models and host weights for you.
7. We'll create proper attribution for you and link to your repo.
8. Once all of this is ready, we will merge into bolts.

After your model or other contribution is in bolts, our team will make sure it maintains compatibility with the other components of the library!

2.1.2 Contribution ideas

Don't have something to contribute? Ping us on [Slack](#) or look at our [Github issues](#)!

We'll help and guide you through the implementation / conversion

2.2 When to use Bolts

2.2.1 For pretrained models

Most bolts have pretrained weights trained on various datasets or algorithms. This is useful when you don't have enough data, time or money to do your own training.

For example, you could use a pretrained VAE to generate features for an image dataset.

```
from pl_bolts.models.autoencoders import VAE
from pl_bolts.models.self_supervised import CPCV2

model1 = VAE(input_height=32, pretrained='imagenet2012')
encoder = model1.encoder
encoder.eval()

# bolts are pretrained on different datasets
model2 = CPCV2(encoder='resnet18', pretrained='imagenet128').freeze()
model3 = CPCV2(encoder='resnet18', pretrained='st110').freeze()
```

```
for (x, y) in own_data:
    features = encoder(x)
    feat2 = model2(x)
    feat3 = model3(x)

# which is better?
```

2.2.2 To finetune on your data

If you have your own data, finetuning can often increase the performance. Since this is pure PyTorch you can use any finetuning protocol you prefer.

Example 1: Unfrozen finetune

```
# unfrozen finetune
model = CPCV2(encoder='resnet18', pretrained='imagenet128')
resnet18 = model.encoder
# don't call .freeze()

classifier = LogisticRegression(...)

for (x, y) in own_data:
    feats = resnet18(x)
    y_hat = classifier(feats)
```

Example 2: Freeze then unfreeze

```
# FREEZE!
model = CPCV2(encoder='resnet18', pretrained='imagenet128')
resnet18 = model.encoder
resnet18.eval()

classifier = LogisticRegression(...)

for epoch in epochs:
    for (x, y) in own_data:
        feats = resnet18(x)
        y_hat = classifier(feats)
        loss = cross_entropy_with_logits(y_hat, y)

    # UNFREEZE after 10 epochs
    if epoch == 10:
        resnet18.unfreeze()
```

2.2.3 For research

Here is where bolts is very different than other libraries with models. It's not just designed for production, but each module is written to be easily extended for research.

```
from pl_bolts.models import ImageGPT
from pl_bolts.self_supervised import SimCLR

class VideoGPT(ImageGPT):
```

(continues on next page)

(continued from previous page)

```

def training_step(self, batch, batch_idx):
    x, y = batch
    x = _shape_input(x)

    logits = self.gpt(x)
    simclr_features = self.simclr(x)

    # -----
    # do something new with GPT logits + simclr_features
    # -----

    loss = self.criterion(logits.view(-1, logits.size(-1)), x.view(-1).long())

    logs = {"loss": loss}
    return {"loss": loss, "log": logs}

```

Or perhaps your research is in self-supervised_learning and you want to do a new SimCLR. In this case, the only thing you want to change is the loss.

By subclassing you can focus on changing a single piece of a system without worrying that the other parts work (because if they are in Bolts, then they do and we've tested it).

```

# subclass SimCLR and change ONLY what you want to try
class ComplexCLR(SimCLR):

    def init_loss(self):
        return self.new_xent_loss

    def new_xent_loss(self):
        out = torch.cat([out_1, out_2], dim=0) n_samples = len(out)

        # Full similarity matrix
        cov = torch.mm(out, out.t().contiguous())
        sim = torch.exp(cov / temperature)

        # Negative similarity
        mask = ~torch.eye(n_samples, device=sim.device).bool()
        neg = sim.masked_select(mask).view(n_samples, -1).sum(dim=-1)

        # -----
        # some new thing we want to do
        # -----

        # Positive similarity :
        pos = torch.exp(torch.sum(out_1 * out_2, dim=-1) / temperature)
        pos = torch.cat([pos, pos], dim=0)
        loss = -torch.log(pos / neg).mean()

    return loss

```

2.3 Callbacks

Callbacks are arbitrary programs which can run at any points in time within a training loop in Lightning.

Bolts houses a collection of callbacks that are community contributed and can work in any Lightning Module!

```
from pl_bolts.callbacks import PrintTableMetricsCallback
import pytorch_lightning as pl

trainer = pl.Trainer(callbacks=[PrintTableMetricsCallback()])
```

2.4 DataModules

In PyTorch, working with data has these major elements.

1. Downloading, saving and preparing the dataset.
2. Splitting into train, val and test.
3. For each split, applying different transforms

A *DataModule* groups together those actions into a single reproducible *DataModule* that can be shared around to guarantee:

1. Consistent data preprocessing (download, splits, etc...)
2. The same exact splits
3. The same exact transforms

```
from pl_bolts.datamodules import ImagenetDataModule

dm = ImagenetDataModule(data_dir=PATH)

# standard PyTorch!
train_loader = dm.train_dataloader()
val_loader = dm.val_dataloader()
test_loader = dm.test_dataloader()

Trainer().fit(
    model,
    train_loader,
    val_loader
)
```

But when paired with PyTorch LightningModules (all bolts models), you can plug and play full dataset definitions with the same splits, transforms, etc...

```
imagenet = ImagenetDataModule(PATH)
model = VAE(datamodule=imagenet)
model = ImageGPT(datamodule=imagenet)
model = GAN(datamodule=imagenet)
```

We even have prebuilt modules to bridge the gap between Numpy, Sklearn and PyTorch

```

from sklearn.datasets import load_boston
from pl_bolts.datamodules import SklearnDataModule

X, y = load_boston(return_X_y=True)
datamodule = SklearnDataModule(X, y)

model = LitModel(datamodule)

```

2.5 Regression Heroes

In case your job or research doesn't need a "hammer", we offer implementations of Classic ML models which benefit from lightning's multi-GPU and TPU support.

So, now you can run huge workloads scalably, without needing to do any engineering. For instance, here we can run logistic Regression on Imagenet (each epoch takes about 3 minutes)!

```

from pl_bolts.models.regression import LogisticRegression

imagenet = ImagenetDataModule(PATH)

# 224 x 224 x 3
pixels_per_image = 150528
model = LogisticRegression(input_dim=pixels_per_image, num_classes=1000)
model.prepare_data = imagenet.prepare_data

trainer = Trainer(gpus=2)
trainer.fit(
    model,
    imagenet.train_dataloader(batch_size=256),
    imagenet.val_dataloader(batch_size=256)
)

```

2.5.1 Linear Regression

Here's an example for Linear regression

```

import pytorch_lightning as pl
from pl_bolts.datamodules import SklearnDataModule
from sklearn.datasets import load_boston

# link the numpy dataset to PyTorch
X, y = load_boston(return_X_y=True)
loaders = SklearnDataModule(X, y)

# training runs training batches while validating against a validation set
model = LinearRegression()
trainer = pl.Trainer(num_gpus=8)
trainer.fit(model, train_dataloader=loaders.train_dataloader(), val_
↳ dataloaders=loaders.val_dataloader())

```

Once you're done, you can run the test set if needed.

```
trainer.test(test_dataloaders=loaders.test_dataloader())
```

But more importantly, you can scale up to many GPUs, TPUs or even CPUs

```
# 8 GPUs
trainer = pl.Trainer(num_gpus=8)

# 8 TPU cores
trainer = pl.Trainer(tpu_cores=8)

# 32 GPUs
trainer = pl.Trainer(num_gpus=8, num_nodes=4)

# 128 CPUs
trainer = pl.Trainer(num_processes=128)
```

2.5.2 Logistic Regression

Here's an example for logistic regression

```
from sklearn.datasets import load_iris
from pl_bolts.models.regression import LogisticRegression
from pl_bolts.datamodules import SklearnDataModule
import pytorch_lightning as pl

# use any numpy or sklearn dataset
X, y = load_iris(return_X_y=True)
dm = SklearnDataModule(X, y, batch_size=12)

# build model
model = LogisticRegression(input_dim=4, num_classes=3)

# fit
trainer = pl.Trainer(tpu_cores=8, precision=16)
trainer.fit(model, train_dataloader=dm.train_dataloader(), val_dataloaders=dm.val_
↳dataloader())

trainer.test(test_dataloaders=dm.test_dataloader())
```

Any input will be flattened across all dimensions except the first one (batch). This means images, sound, etc... work out of the box.

```
# create dataset
dm = MNISTDataModule(num_workers=0, data_dir=tmpdir)

model = LogisticRegression(input_dim=28 * 28, num_classes=10, learning_rate=0.001)
model.prepare_data = dm.prepare_data
model.train_dataloader = dm.train_dataloader
model.val_dataloader = dm.val_dataloader
model.test_dataloader = dm.test_dataloader

trainer = pl.Trainer(max_epochs=2)
trainer.fit(model)
trainer.test(model)
# {test_acc: 0.92}
```

But more importantly, you can scale up to many GPUs, TPUs or even CPUs

```
# 8 GPUs
trainer = pl.Trainer(num_gpus=8)

# 8 TPUs
trainer = pl.Trainer(tpu_cores=8)

# 32 GPUs
trainer = pl.Trainer(num_gpus=8, num_nodes=4)

# 128 CPUs
trainer = pl.Trainer(num_processes=128)
```

2.6 Regular PyTorch

Everything in bolts also works with regular PyTorch since they are all just `nn.Modules`!

However, if you train using Lightning you don't have to deal with engineering code :)

2.7 Command line support

Any bolt module can also be trained from the command line

```
cd pl_bolts/models/autoencoders/basic_vae
python basic_vae_pl_module.py
```

Each script accepts Argparse arguments for both the lightning trainer and the model

```
python basic_vae_pl_module.py --latent_dim 32 --batch_size 32 --gpus 4 --max_epochs 12
```


MODEL QUALITY CONTROL

For bolts to be added to the library we have a **rigorous** quality control checklist

3.1 Bolts vs my own repo

We hope you keep your own repo still! We want to link to it to let people know. However, by adding your contribution to bolts you get these **additional** benefits!

1. More visibility! (more people all over the world use your code)
2. We test your code on every PR (CPUs, GPUs, TPUs).
3. We host the docs (and test on every PR).
4. We help you build thorough, beautiful documentation.
5. We help you build robust tests.
6. We'll pretrain expensive models for you and host weights.
7. We will improve the speed of your models!
8. Eligible for invited talks to discuss your implementation.
9. Lightning swag + involvement in the broader contributor community :)

Note: You still get to keep your attribution and be recognized for your work!

Note: Bolts is a community library built by incredible people like you!

3.2 Contribution requirements

3.2.1 Benchmarked

Models have known performance results on common baseline datasets.

3.2.2 Device agnostic

Models must work on CPUs, GPUs and TPUs without changing code. We help authors with this.

```
# bad
encoder.to(device)
```

3.2.3 Fast

We inspect models for computational inefficiencies and help authors meet the bar. Granted, sometimes the approaches are slow for mathematical reasons. But anything related to engineering we help overcome.

```
# bad
mtx = ...
for xi in rows:
    for yi in cols:
        mxt[xi, yi] = ...

# good
x = x.item().numpy()
x = np.some_fx(x)
x = torch.tensor(x)
```

3.2.4 Tested

Models are tested on every PR (on CPUs, GPUs and soon TPUs).

- Live build
- Tests

3.2.5 Modular

Models are modularized to be extended and reused easily.

```
# GOOD!
class LitVAE(pl.LightningModule):

    def init_prior(self, ...):
        # enable users to override interesting parts of each model

    def init_posterior(self, ...):
        # enable users to override interesting parts of each model

# BAD
class LitVAE(pl.LightningModule):

    def __init__(self):
        self.prior = ...
        self.posterior = ...
```

3.2.6 Attribution

Any models and weights that are contributed are attributed to you as the author(s).

We request that each contribution have:

- The original paper link
- The list of paper authors
- The link to the original paper code (if available)
- The link to your repo
- Your name and your team's name as the implementation authors.
- Your team's affiliation
- Any generated examples, or result plots.
- Hyperparameter configurations for the results.

Thank you for all your amazing contributions!

3.3 The bar seems high

If your model doesn't yet meet this bar, no worries! Please open the PR and our team of core contributors will help you get there!

3.4 Do you have contribution ideas?

Yes! Check the Github issues for requests from the Lightning team and the community! We'll even work with you to finish your implementation! Then we'll help you pretrain it and cover the compute costs when possible.

BUILD A CALLBACK

This module houses a collection of callbacks that can be passed into the trainer

```
from pl_bolts.callbacks import PrintTableMetricsCallback

trainer = pl.Trainer(callbacks=[PrintTableMetricsCallback()])

# loss|train_loss|val_loss|epoch
# -----
# 2.2541470527648926|2.2541470527648926|2.2158432006835938|0
```

4.1 What is a Callback

A callback is a self-contained program that can be intertwined into a training pipeline without polluting the main research logic.

4.2 Create a Callback

Creating a callback is simple:

```
from pytorch_lightning.callbacks import Callback

class MyCallback(Callback):
    def on_epoch_end(self, trainer, pl_module):
        # do something
```

Please refer to [Callback docs](#) for a full list of the 20+ hooks available.

INFO CALLBACKS

These callbacks give all sorts of useful information during training.

5.1 Print Table Metrics

This callback prints training metrics to a table. It's very bare-bones for speed purposes.

class `pl_bolts.callbacks.printing.PrintTableMetricsCallback`

Bases: `pytorch_lightning.callbacks.Callback`

Prints a table with the metrics in columns on every epoch end

Example:

```
from pl_bolts.callbacks import PrintTableMetricsCallback

callback = PrintTableMetricsCallback()
```

Pass into trainer like so:

```
trainer = pl.Trainer(callbacks=[callback])
trainer.fit(...)

# -----
# at the end of every epoch it will print
# -----

# loss|train_loss|val_loss|epoch
# -----
# 2.2541470527648926|2.2541470527648926|2.2158432006835938|0
```

5.2 Data Monitoring in LightningModule

The data monitoring callbacks allow you to log and inspect the distribution of data that passes through the training step and layers of the model. When used in combination with a supported logger, the `TrainingDataMonitor` creates a histogram for each *batch* input in `training_step()` and sends it to the logger:

```
from pl_bolts.callbacks import TrainingDataMonitor
from pytorch_lightning import Trainer

# log the histograms of input data sent to LightningModule.training_step
monitor = TrainingDataMonitor(log_every_n_steps=25)

model = YourLightningModule()
trainer = Trainer(callbacks=[monitor])
trainer.fit()
```

The second, more advanced `ModuleDataMonitor` callback tracks histograms for the data that passes through the model itself and its submodules, i.e., it tracks all `.forward()` calls and registers the in- and outputs. You can track all or just a selection of submodules:

```
from pl_bolts.callbacks import ModuleDataMonitor
from pytorch_lightning import Trainer

# log the in- and output histograms of LightningModule's `forward`
monitor = ModuleDataMonitor()

# all submodules in LightningModule
monitor = ModuleDataMonitor(submodules=True)

# specific submodules
monitor = ModuleDataMonitor(submodules=["generator", "generator.conv1"])

model = YourLightningModule()
trainer = Trainer(callbacks=[monitor])
trainer.fit()
```

This is especially useful for debugging the data flow in complex models and to identify numerical instabilities.

5.3 Model Verification

5.3.1 Gradient-Check for Batch-Optimization

Gradient descent over a batch of samples can not only benefit the optimization but also leverages data parallelism. However, one has to be careful not to mix data across the batch dimension. Only a small error in a reshape or permutation operation results in the optimization getting stuck and you won't even get a runtime error. How can one tell if the model mixes data in the batch? A simple trick is to do the following:

1. run the model on an example batch (can be random data)
2. get the output batch and select the *n*-th sample (choose *n*)
3. compute a dummy loss value of only that sample and compute the gradient w.r.t the entire input batch
4. observe that only the *i*-th sample in the input batch has non-zero gradient

If the gradient is non-zero for the other samples in the batch, it means the forward pass of the model is mixing data! The *BatchGradientVerificationCallback* does all of that for you before training begins.

```
from pytorch_lightning import Trainer
from pl_bolts.callbacks import BatchGradientVerificationCallback

model = YourLightningModule()
verification = BatchGradientVerificationCallback()
trainer = Trainer(callbacks=[verification])
trainer.fit(model)
```

This Callback will warn the user with the following message in case data mixing inside the batch is detected:

```
Your model is mixing data across the batch dimension.
This can lead to wrong gradient updates in the optimizer.
Check the operations that reshape and permute tensor dimensions in your model.
```

A non-Callback version *BatchGradientVerification* that works with any PyTorch *Module* is also available:

```
from pl_bolts.utils import BatchGradientVerification

model = YourPyTorchModel()
verification = BatchGradientVerification(model)
valid = verification.check(input_array=torch.rand(2, 3, 4), sample_idx=1)
```

In this example we run the test on a batch size 2 by inspecting gradients on the second sample.

SELF-SUPERVISED CALLBACKS

Useful callbacks for self-supervised learning models

6.1 BYOLMAWeightUpdate

The exponential moving average weight-update rule from Bootstrap Your Own Latent (BYOL).

```
class pl_bolts.callbacks.byol_updates.BYOLMAWeightUpdate (initial_tau=0.996)  
    Bases: pytorch_lightning.Callback
```

Weight update rule from BYOL.

Your model should have:

- `self.online_network`
- `self.target_network`

Updates the `target_network` params using an exponential moving average update rule weighted by `tau`. BYOL claims this keeps the `online_network` from collapsing.

Note: Automatically increases `tau` from `initial_tau` to 1.0 with every training step

Example:

```
# model must have 2 attributes  
model = Model()  
model.online_network = ...  
model.target_network = ...  
  
trainer = Trainer(callbacks=[BYOLMAWeightUpdate()])
```

Parameters `initial_tau` (float) – starting tau. Auto-updates with every training step

6.2 SSLOnlineEvaluator

Appends a MLP for fine-tuning to the given model. Callback has its own mini-inner loop.

```
class pl_bolts.callbacks.ssl_online.SSLOnlineEvaluator(dataset,          drop_p=0.2,  
                                                    hidden_dim=None,  
                                                    z_dim=None,  
                                                    num_classes=None)
```

Bases: `pytorch_lightning.Callback`

Attaches a MLP for fine-tuning using the standard self-supervised protocol.

Example:

```
# your model must have 2 attributes  
model = Model()  
model.z_dim = ... # the representation dim  
model.num_classes = ... # the num of classes in the model  
  
online_eval = SSLOnlineEvaluator(  
    z_dim=model.z_dim,  
    num_classes=model.num_classes,  
    dataset='imagenet'  
)
```

Parameters

- **dataset** `//` (`str`) – if stl10, need to get the labeled batch
- **drop_p** `//` (`float`) – Dropout probability
- **hidden_dim** `//` (`Optional[int]`) – Hidden dimension for the fine-tune MLP
- **z_dim** `//` (`Optional[int]`) – Representation dimension
- **num_classes** `//` (`Optional[int]`) – Number of classes

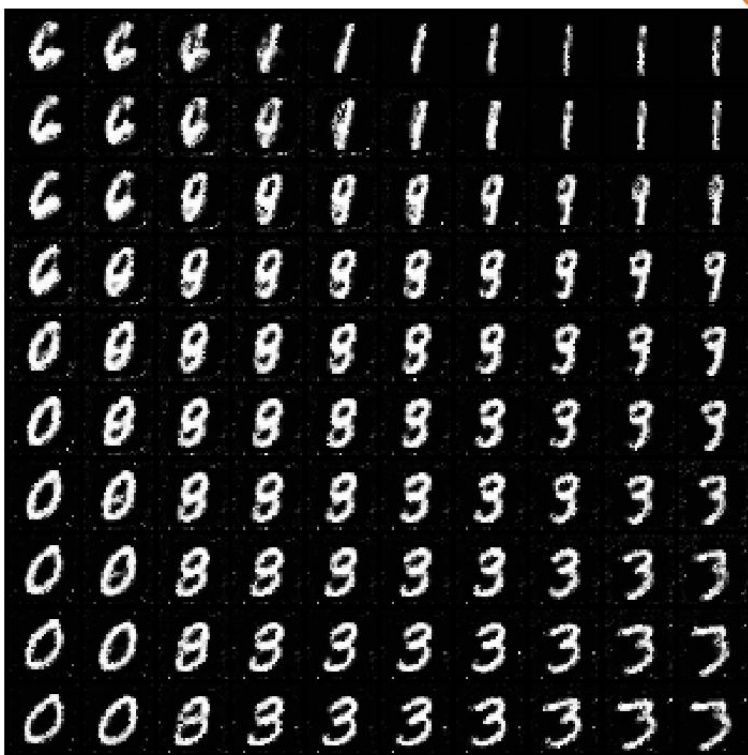
VARIATIONAL CALLBACKS

Useful callbacks for GANs, variational-autoencoders or anything with latent spaces.

7.1 Latent Dim Interpolator

Interpolates latent dims.

Example output:



```
class pl_bolts.callbacks.variational.LatentDimInterpolator(interpolate_epoch_interval=20,  
                                                         range_start=-5,  
                                                         range_end=5,  
                                                         steps=11,  
                                                         num_samples=2,  
                                                         normalize=True)
```

Bases: `pytorch_lightning.callbacks.Callback`

Interpolates the latent space for a model by setting all dims to zero and stepping through the first two dims increasing one unit at a time.

Default interpolates between `[-5, 5]` `(-5, -4, -3, ..., 3, 4, 5)`

Example:

```
from pl_bolts.callbacks import LatentDimInterpolator

Trainer(callbacks=[LatentDimInterpolator()])
```

Parameters

- **`interpolate_epoch_interval`** `(int)` – default 20
- **`range_start`** `(int)` – default -5
- **`range_end`** `(int)` – default 5
- **`steps`** `(int)` – number of step between start and end
- **`num_samples`** `(int)` – default 2
- **`normalize`** `(bool)` – default True (change image to (0, 1) range)

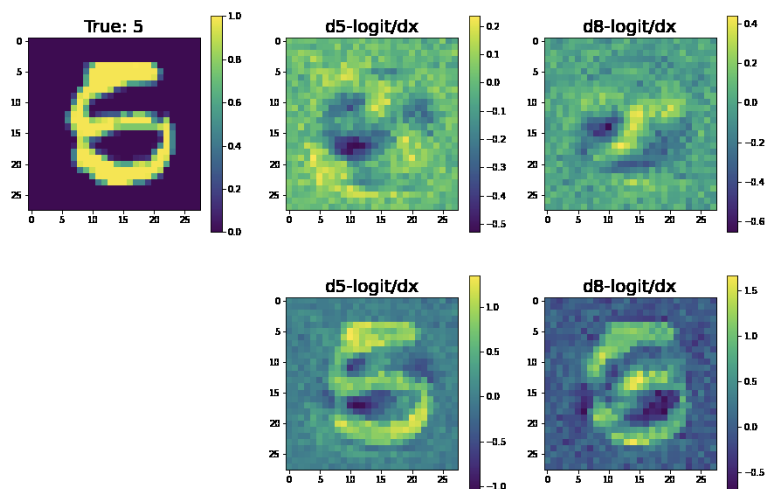
VISION CALLBACKS

Useful callbacks for vision models

8.1 Confused Logit

Shows how the input would have to change to move the prediction from one logit to the other

Example outputs:



```
class pl_bolts.callbacks.vision.confused_logit.ConfusedLogitCallback(top_k,  
                                                                    projec-  
                                                                    tion_factor=3,  
                                                                    min_logit_value=5.0,  
                                                                    log-  
                                                                    ging_batch_interval=20,  
                                                                    max_logit_difference=0.1)
```

Bases: `pytorch_lightning.Callback`

Takes the logit predictions of a model and when the probabilities of two classes are very close, the model doesn't have high certainty that it should pick one vs the other class.

This callback shows how the input would have to change to swing the model from one label prediction to the other.

In this case, the network predicts a 5... but gives almost equal probability to an 8. The images show what about the original 5 would have to change to make it more like a 5 or more like an 8.

For each confused logit the confused images are generated by taking the gradient from a logit wrt an input for the top two closest logits.

Example:

```
from pl_bolts.callbacks.vision import ConfusedLogitCallback
trainer = Trainer(callbacks=[ConfusedLogitCallback()])
```

Note: Whenever called, this model will look for `self.last_batch` and `self.last_logits` in the `LightningModule`.

Note: This callback supports tensorboard only right now.

Authored by:

- Alfredo Canziani

Parameters

- `top_k` (int) – How many “offending” images we should plot
 - `projection_factor` (int) – How much to multiply the input image to make it look more like this logit label
 - `min_logit_value` (float) – Only consider logit values above this threshold
 - `logging_batch_interval` (int) – How frequently to inspect/potentially plot something
 - `max_logit_difference` (float) – When the top 2 logits are within this threshold we consider them confused
-

8.2 Tensorboard Image Generator

Generates images from a generative model and plots to tensorboard

class `pl_bolts.callbacks.vision.image_generation.TensorboardGenerativeModelImageSampler` (num

Bases: `pytorch_lightning.Callback`

Generates images and logs to tensorboard. Your model must implement the `forward` function for generation

Requirements:


```
# model must have img_dim arg
model.img_dim = (1, 28, 28)

# model forward must work for sampling
z = torch.rand(batch_size, latent_dim)
img_samples = your_model(z)
```

Example:

```
from pl_bolts.callbacks import TensorboardGenerativeModelImageSampler

trainer = Trainer(callbacks=[TensorboardGenerativeModelImageSampler()])
```

Parameters

- **num_samples** (int) – Number of images displayed in the grid. Default: 3.
- **nrow** (int) – Number of images displayed in each row of the grid. The final grid size is (B / nrow, nrow). Default: 8.
- **padding** (int) – Amount of padding. Default: 2.
- **normalize** (bool) – If True, shift the image to the range (0, 1), by the min and max values specified by range. Default: False.
- **norm_range** (Optional[Tuple[int, int]]) – Tuple (min, max) where min and max are numbers, then these numbers are used to normalize the image. By default, min and max are computed from the tensor.
- **scale_each** (bool) – If True, scale each image in the batch of images separately rather than the (min, max) over all images. Default: False.
- **pad_value** (int) – Value for the padded pixels. Default: 0.

DATAMODULES

DataModules (introduced in PyTorch Lightning 0.9.0) decouple the data from a model. A DataModule is simply a collection of a training dataloader, val dataloader and test dataloader. In addition, it specifies how to:

- Download/prepare data.
- Train/val/test splits.
- Transform

Then you can use it like this:

Example:

```
dm = MNISTDataModule('path/to/data')
model = LitModel()

trainer = Trainer()
trainer.fit(model, datamodule=dm)
```

Or use it manually with plain PyTorch

Example:

```
dm = MNISTDataModule('path/to/data')
for batch in dm.train_dataloader():
    ...
for batch in dm.val_dataloader():
    ...
for batch in dm.test_dataloader():
    ...
```

Please visit the PyTorch Lightning documentation for more details on DataModules

SKLEARN DATAMODULE

Utilities to map sklearn or numpy datasets to PyTorch Dataloaders with automatic data splits and GPU/TPU support.

```
from sklearn.datasets import load_boston
from pl_bolts.datamodules import SklearnDataModule

X, y = load_boston(return_X_y=True)
loaders = SklearnDataModule(X, y)

train_loader = loaders.train_dataloader(batch_size=32)
val_loader = loaders.val_dataloader(batch_size=32)
test_loader = loaders.test_dataloader(batch_size=32)
```

Or build your own torch datasets

```
from sklearn.datasets import load_boston
from pl_bolts.datamodules import SklearnDataset

X, y = load_boston(return_X_y=True)
dataset = SklearnDataset(X, y)
loader = DataLoader(dataset)
```

10.1 Sklearn Dataset Class

Transforms a sklearn or numpy dataset to a PyTorch Dataset.

```
class pl_bolts.datamodules.sklearn_datamodule.SklearnDataset(X, y,
                                                             X_transform=None,
                                                             y_transform=None)
```

Bases: `torch.utils.data.Dataset`

Mapping between numpy (or sklearn) datasets to PyTorch datasets.

Example

```
>>> from sklearn.datasets import load_boston
>>> from pl_bolts.datamodules import SklearnDataset
...
>>> X, y = load_boston(return_X_y=True)
>>> dataset = SklearnDataset(X, y)
>>> len(dataset)
506
```

Parameters

- **X** (ndarray) – Numpy ndarray
 - **y** (ndarray) – Numpy ndarray
 - **X_transform** (Optional[Any]) – Any transform that works with Numpy arrays
 - **y_transform** (Optional[Any]) – Any transform that works with Numpy arrays
-

10.2 Sklearn DataModule Class

Automatically generates the train, validation and test splits for a Numpy dataset. They are set up as dataloaders for convenience. Optionally, you can pass in your own validation and test splits.

```
class pl_bolts.datamodules.sklearn_datamodule.SklearnDataModule(X, y,
                                                                x_val=None,
                                                                y_val=None,
                                                                x_test=None,
                                                                y_test=None,
                                                                val_split=0.2,
                                                                test_split=0.1,
                                                                num_workers=2,
                                                                ran-
                                                                dom_state=1234,
                                                                shuffle=True,
                                                                batch_size=16,
                                                                pin_memory=False,
                                                                drop_last=False,
                                                                *args,
                                                                **kwargs)
```

Bases: `pytorch_lightning.LightningDataModule`

Automatically generates the train, validation and test splits for a Numpy dataset. They are set up as dataloaders for convenience. Optionally, you can pass in your own validation and test splits.

Example

```
>>> from sklearn.datasets import load_boston
>>> from pl_bolts.datamodules import SklearnDataModule
...
>>> X, y = load_boston(return_X_y=True)
>>> loaders = SklearnDataModule(X, y, batch_size=32)
...
>>> # train set
>>> train_loader = loaders.train_dataloader()
>>> len(train_loader.dataset)
355
>>> len(train_loader)
12
>>> # validation set
>>> val_loader = loaders.val_dataloader()
>>> len(val_loader.dataset)
100
>>> len(val_loader)
4
>>> # test set
>>> test_loader = loaders.test_dataloader()
>>> len(test_loader.dataset)
51
>>> len(test_loader)
2
```


VISION DATAMODULES

The following are pre-built datamodules for computer-vision.

11.1 Supervised learning

These are standard vision datasets with the train, test, val splits pre-generated in DataLoaders with the standard transforms (and Normalization) values

11.1.1 BinaryMNIST

```
class pl_bolts.datamodules.binary_mnist_datamodule.BinaryMNISTDataModule (data_dir=None,  
val_split=0.2,  
num_workers=16,  
normalize=False,  
batch_size=32,  
seed=42,  
shuffle=False,  
pin_memory=False,  
drop_last=False,  
*args,  
**kwargs)
```

Bases: `pl_bolts.datamodules.vision_datamodule.VisionDataModule`

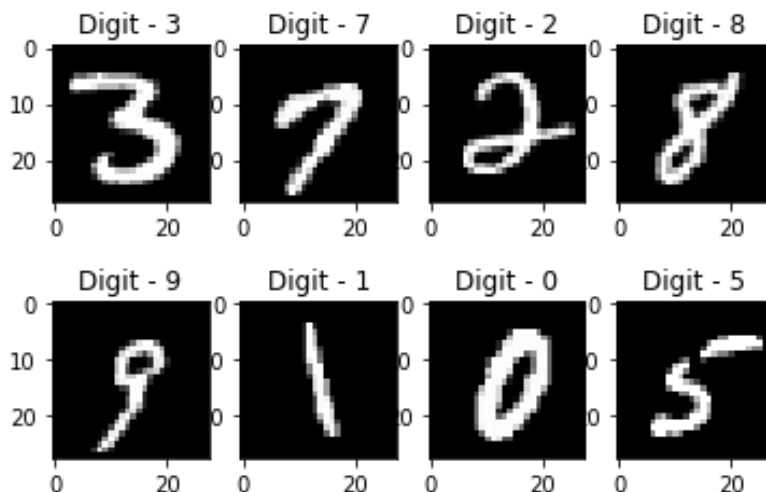
Specs:

- 10 classes (1 per digit)
- Each image is (1 x 28 x 28)

Binary MNIST, train, val, test splits and transforms

Transforms:

```
mnist_transforms = transform_lib.Compose([  
    transform_lib.ToTensor()  
])
```



Example:

```
from pl_bolts.datamodules import BinaryMNISTDataModule

dm = BinaryMNISTDataModule('.')
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (Optional[str]) – Where to save/load the data
- **val_split** (Union[int, float]) – Percent (float) or number (int) of samples to use for the validation split
- **num_workers** (int) – How many workers to use for loading data
- **normalize** (bool) – If true applies image normalize
- **batch_size** (int) – How many samples per batch to load
- **seed** (int) – Random seed to be used for train/val/test splits
- **shuffle** (bool) – If true shuffles the train data every epoch
- **pin_memory** (bool) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (bool) – If true drops the last incomplete batch

default_transforms ()

Default transform for the dataset

Return type Callable

property num_classes

Return: 10

Return type int

11.1.2 CityScapes

```
class pl_bolts.datamodules.cityscapes_datamodule.CityscapesDataModule(data_dir,
                                                                    quality_mode='fine',
                                                                    target_type='instance',
                                                                    num_workers=16,
                                                                    batch_size=32,
                                                                    seed=42,
                                                                    shuffle=False,
                                                                    pin_memory=False,
                                                                    drop_last=False,
                                                                    *args,
                                                                    **kwargs)
```

Bases: `pytorch_lightning.LightningDataModule`



Standard Cityscapes, train, val, test splits and transforms

Note: You need to have downloaded the Cityscapes dataset first and provide the path to where it is saved.

You can download the dataset here: <https://www.cityscapes-dataset.com/>

Specs:

- 30 classes (road, person, sidewalk, etc...)
- (image, target) - image dims: (3 x 1024 x 2048), target dims: (1024 x 2048)

Transforms:

```
transforms = transform_lib.Compose([
    transform_lib.ToTensor(),
    transform_lib.Normalize(
        mean=[0.28689554, 0.32513303, 0.28389177],
        std=[0.18696375, 0.19017339, 0.18720214]
    )
])
```

Example:

```
from pl_bolts.datamodules import CityscapesDataModule

dm = CityscapesDataModule(PATH)
```

(continues on next page)

(continued from previous page)

```
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Or you can set your own transforms

Example:

```
dm.train_transforms = ...
dm.test_transforms = ...
dm.val_transforms = ...
dm.target_transforms = ...
```

Parameters

- **data_dir** (str) – where to load the data from path, i.e. where directory leftImg8bit and gtFine or gtCoarse are located
- **quality_mode** (str) – the quality mode to use, either ‘fine’ or ‘coarse’
- **target_type** (str) – targets to use, either ‘instance’ or ‘semantic’
- **num_workers** (int) – how many workers to use for loading data
- **batch_size** (int) – number of examples per training/eval step
- **seed** (int) – random seed to be used for train/val/test splits
- **shuffle** (bool) – If true shuffles the data every epoch
- **pin_memory** (bool) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (bool) – If true drops the last incomplete batch

test_dataloader()

Cityscapes test set

Return type DataLoader

train_dataloader()

Cityscapes train set

Return type DataLoader

val_dataloader()

Cityscapes val set

Return type DataLoader

property num_classes

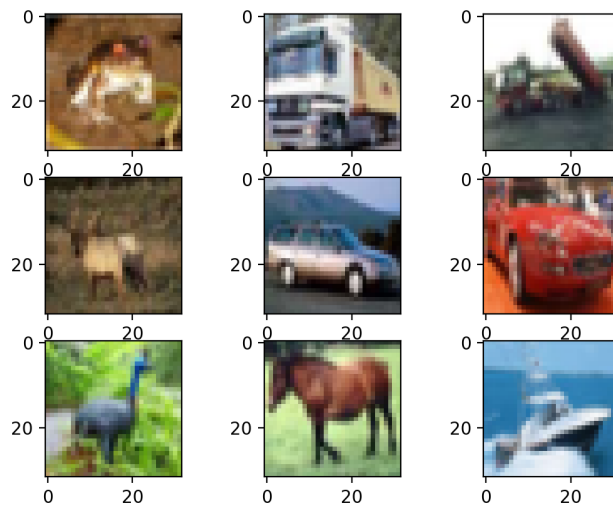
Return: 30

Return type int

11.1.3 CIFAR-10

```
class pl_bolts.datamodules.cifar10_datamodule.CIFAR10DataModule(data_dir=None,
                                                             val_split=0.2,
                                                             num_workers=16,
                                                             normal-
                                                             ize=False,
                                                             batch_size=32,
                                                             seed=42,
                                                             shuffle=False,
                                                             pin_memory=False,
                                                             drop_last=False,
                                                             *args,
                                                             **kwargs)
```

Bases: `pl_bolts.datamodules.vision_datamodule.VisionDataModule`



Specs:

- 10 classes (1 per class)
- Each image is (3 x 32 x 32)

Standard CIFAR10, train, val, test splits and transforms

Transforms:

```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor(),
    transforms.Normalize(
        mean=[x / 255.0 for x in [125.3, 123.0, 113.9]],
        std=[x / 255.0 for x in [63.0, 62.1, 66.7]]
    )
])
```

Example:

```
from pl_bolts.datamodules import CIFAR10DataModule

dm = CIFAR10DataModule(PATH)
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Or you can set your own transforms

Example:

```
dm.train_transforms = ...
dm.test_transforms = ...
dm.val_transforms = ...
```

Parameters

- **data_dir** (Optional[str]) – Where to save/load the data
- **val_split** (Union[int, float]) – Percent (float) or number (int) of samples to use for the validation split
- **num_workers** (int) – How many workers to use for loading data
- **normalize** (bool) – If true applies image normalize
- **batch_size** (int) – How many samples per batch to load
- **seed** (int) – Random seed to be used for train/val/test splits
- **shuffle** (bool) – If true shuffles the train data every epoch
- **pin_memory** (bool) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (bool) – If true drops the last incomplete batch

default_transforms ()

Default transform for the dataset

Return type `Callable`

property num_classes

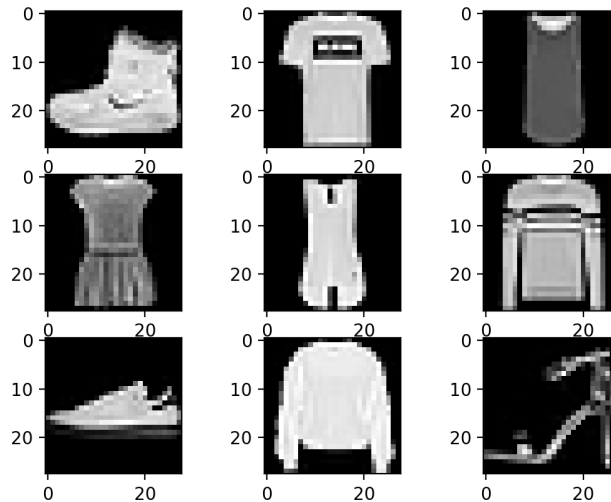
Return: 10

Return type `int`

11.1.4 FashionMNIST

```
class pl_bolts.datamodules.fashion_mnist_datamodule.FashionMNISTDataModule(data_dir=None,
                                                                           val_split=0.2,
                                                                           num_workers=16,
                                                                           normal-
                                                                           ize=False,
                                                                           batch_size=32,
                                                                           seed=42,
                                                                           shuf-
                                                                           fle=False,
                                                                           pin_memory=False,
                                                                           drop_last=False,
                                                                           *args,
                                                                           **kwargs)
```

Bases: `pl_bolts.datamodules.vision_datamodule.VisionDataModule`



Specs:

- 10 classes (1 per type)
- Each image is (1 x 28 x 28)

Standard FashionMNIST, train, val, test splits and transforms

Transforms:

```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor()
])
```

Example:

```
from pl_bolts.datamodules import FashionMNISTDataModule

dm = FashionMNISTDataModule('.')
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (Optional[str]) – Where to save/load the data
- **val_split** (Union[int, float]) – Percent (float) or number (int) of samples to use for the validation split
- **num_workers** (int) – How many workers to use for loading data
- **normalize** (bool) – If true applies image normalize
- **batch_size** (int) – How many samples per batch to load
- **seed** (int) – Random seed to be used for train/val/test splits
- **shuffle** (bool) – If true shuffles the train data every epoch
- **pin_memory** (bool) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (bool) – If true drops the last incomplete batch

default_transforms ()

Default transform for the dataset

Return type Callable

property num_classes

Return: 10

Return type int

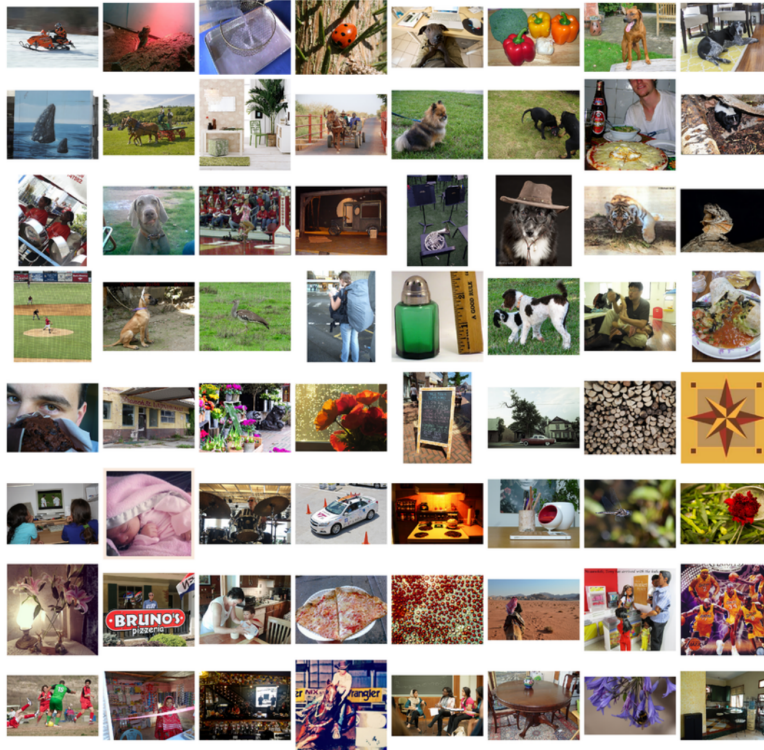
11.1.5 Imagenet

```
class pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule (data_dir,
                                                                    meta_dir=None,
                                                                    num_imgs_per_val_class=50,
                                                                    im-
                                                                    age_size=224,
                                                                    num_workers=16,
                                                                    batch_size=32,
                                                                    shuf-
                                                                    fle=False,
                                                                    pin_memory=False,
                                                                    drop_last=False,
                                                                    *args,
                                                                    **kwargs)
```

Bases: pytorch_lightning.LightningDataModule

Specs:

- 1000 classes



- Each image is (3 x varies x varies) (here we default to 3 x 224 x 224)

Imagenet train, val and test dataloaders.

The train set is the imagenet train.

The val set is taken from the train set with `num_imgs_per_val_class` images per class. For example if `num_imgs_per_val_class=2` then there will be 2,000 images in the validation set.

The test set is the official imagenet validation set.

Example:

```
from pl_bolts.datamodules import ImagenetDataModule

dm = ImagenetDataModule(IMGNET_PATH)
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Parameters

- `data_dir` (str) – path to the imagenet dataset file
- `meta_dir` (Optional[str]) – path to meta.bin file
- `num_imgs_per_val_class` (int) – how many images per class for the validation set
- `image_size` (int) – final image size
- `num_workers` (int) – how many data workers
- `batch_size` (int) – batch_size

- **shuffle** (bool) – If true shuffles the data every epoch
- **pin_memory** (bool) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (bool) – If true drops the last incomplete batch

prepare_data()

This method already assumes you have imagenet2012 downloaded. It validates the data using the meta.bin.

Warning: Please download imagenet on your own first.

Return type None

test_dataloader()

Uses the validation split of imagenet2012 for testing

Return type `DataLoader`

train_dataloader()

Uses the train split of imagenet2012 and puts away a portion of it for the validation split

Return type `DataLoader`

train_transform()

The standard imagenet transforms

```
transform_lib.Compose([
    transform_lib.RandomResizedCrop(self.image_size),
    transform_lib.RandomHorizontalFlip(),
    transform_lib.ToTensor(),
    transform_lib.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    ),
])
```

Return type `Callable`

val_dataloader()

Uses the part of the train split of imagenet2012 that was not used for training via *num_imgs_per_val_class*

Parameters

- **batch_size** – the batch size
- **transforms** – the transforms

Return type `DataLoader`

val_transform()

The standard imagenet transforms for validation

```
transform_lib.Compose([
    transform_lib.Resize(self.image_size + 32),
    transform_lib.CenterCrop(self.image_size),
    transform_lib.ToTensor(),
    transform_lib.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )
])
```

(continues on next page)

(continued from previous page)

```

    ),
    ])
```

Return type `Callable`**property** `num_classes`

Return:

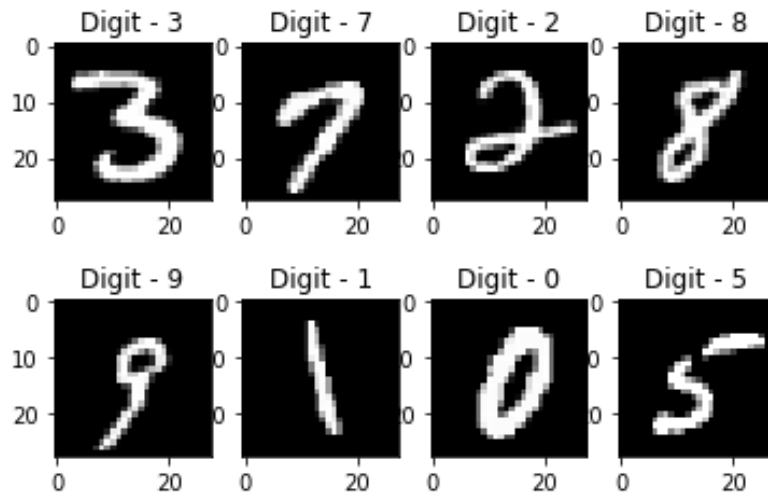
1000

Return type `int`

11.1.6 MNIST

```

class pl_bolts.datamodules.mnist_datamodule.MNISTDataModule(
    data_dir=None,
    val_split=0.2,
    num_workers=16,
    normalize=False,
    batch_size=32,
    seed=42,
    shuffle=False,
    pin_memory=False,
    drop_last=False,
    *args, **kwargs
)
```

Bases: `pl_bolts.datamodules.vision_datamodule.VisionDataModule`**Specs:**

- 10 classes (1 per digit)
- Each image is (1 x 28 x 28)

Standard MNIST, train, val, test splits and transforms

Transforms:

```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor()
])
```

Example:

```
from pl_bolts.datamodules import MNISTDataModule

dm = MNISTDataModule('.')
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (Optional[str]) – Where to save/load the data
- **val_split** (Union[int, float]) – Percent (float) or number (int) of samples to use for the validation split
- **num_workers** (int) – How many workers to use for loading data
- **normalize** (bool) – If true applies image normalize
- **batch_size** (int) – How many samples per batch to load
- **seed** (int) – Random seed to be used for train/val/test splits
- **shuffle** (bool) – If true shuffles the train data every epoch
- **pin_memory** (bool) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (bool) – If true drops the last incomplete batch

default_transforms ()

Default transform for the dataset

Return type Callable

property num_classes

Return: 10

Return type int

11.2 Semi-supervised learning

The following datasets have support for unlabeled training and semi-supervised learning where only a few examples are labeled.

11.2.1 Imagenet (ssl)

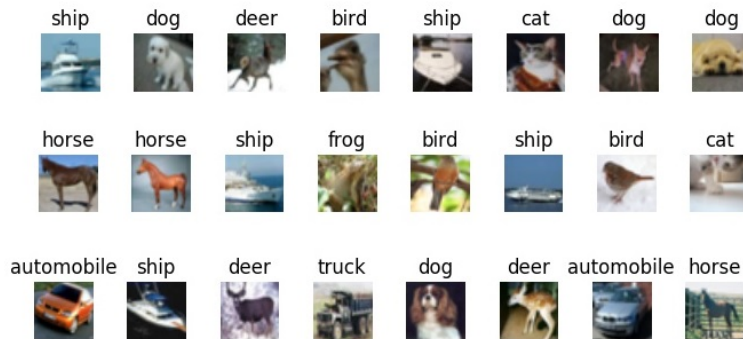
```
class pl_bolts.datamodules.ssl_imagenet_datamodule.SSLImagenetDataModule(data_dir,
                                                                    meta_dir=None,
                                                                    num_workers=16,
                                                                    batch_size=32,
                                                                    shuffle=False,
                                                                    pin_memory=False,
                                                                    drop_last=False,
                                                                    *args,
                                                                    **kwargs)
```

Bases: `pytorch_lightning.LightningDataModule`

11.2.2 STL-10

```
class pl_bolts.datamodules.stl10_datamodule.STL10DataModule(data_dir=None,
                                                             unlabeled_val_split=5000,
                                                             train_val_split=500,
                                                             num_workers=16,
                                                             batch_size=32,
                                                             seed=42, shuffle=False,
                                                             pin_memory=False,
                                                             drop_last=False,
                                                             *args, **kwargs)
```

Bases: `pytorch_lightning.LightningDataModule`



Specs:

- 10 classes (1 per type)
- Each image is (3 x 96 x 96)

Standard STL-10, train, val, test splits and transforms. STL-10 has support for doing validation splits on the labeled or unlabeled splits

Transforms:

```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor(),
    transforms.Normalize(
        mean=(0.43, 0.42, 0.39),
        std=(0.27, 0.26, 0.27)
    )
])
```

Example:

```
from pl_bolts.datamodules import STL10DataModule

dm = STL10DataModule(PATH)
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (Optional[str]) – where to save/load the data
- **unlabeled_val_split** (int) – how many images from the unlabeled training split to use for validation
- **train_val_split** (int) – how many images from the labeled training split to use for validation
- **num_workers** (int) – how many workers to use for loading data
- **batch_size** (int) – the batch size
- **seed** (int) – random seed to be used for train/val/test splits
- **shuffle** (bool) – If true shuffles the data every epoch
- **pin_memory** (bool) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (bool) – If true drops the last incomplete batch

prepare_data()

Downloads the unlabeled, train and test split

Return type None

test_dataloader()

Loads the test split of STL10

Parameters

- **batch_size** – the batch size
- **transforms** – the transforms

Return type DataLoader

train_dataloader()

Loads the ‘unlabeled’ split minus a portion set aside for validation via *unlabeled_val_split*.

Return type DataLoader

train_dataloader_mixed()

Loads a portion of the ‘unlabeled’ training data and ‘train’ (labeled) data. both portions have a subset removed for validation via *unlabeled_val_split* and *train_val_split*

Parameters

- **batch_size** – the batch size
- **transforms** – a sequence of transforms

Return type `DataLoader`**val_dataloader()**

Loads a portion of the ‘unlabeled’ training data set aside for validation The val dataset = (unlabeled - train_val_split)

Parameters

- **batch_size** – the batch size
- **transforms** – a sequence of transforms

Return type `DataLoader`**val_dataloader_mixed()**

Loads a portion of the ‘unlabeled’ training data set aside for validation along with the portion of the ‘train’ dataset to be used for validation

unlabeled_val = (unlabeled - train_val_split)

labeled_val = (train- train_val_split)

full_val = unlabeled_val + labeled_val

Parameters

- **batch_size** – the batch size
- **transforms** – a sequence of transforms

Return type `DataLoader`

DATASETS

Collection of useful datasets

12.1 Debugging

Use these datasets to debug

12.1.1 DummyDataset

class `pl_bolts.datasets.dummy_dataset.DummyDataset` (**shapes, num_samples=10000*)
Bases: `torch.utils.data.Dataset`
Generate a dummy dataset

Example

```
>>> from pl_bolts.datasets import DummyDataset
>>> from torch.utils.data import DataLoader
>>> # mnist dims
>>> ds = DummyDataset((1, 28, 28), (1, ))
>>> dl = DataLoader(ds, batch_size=7)
>>> # get first batch
>>> batch = next(iter(dl))
>>> x, y = batch
>>> x.size()
torch.Size([7, 1, 28, 28])
>>> y.size()
torch.Size([7, 1])
```

Parameters

- ***shapes** – list of shapes
- **num_samples** (`int`) – how many samples to use in this dataset

12.1.2 DummyDetectionDataset

```
class pl_bolts.datasets.dummy_dataset.DummyDetectionDataset (img_shape=(3, 256,  
                                                             256), num_boxes=1,  
                                                             num_classes=2,  
                                                             num_samples=10000)
```

Bases: `torch.utils.data.Dataset`

Generate a dummy dataset for detection

Example

```
>>> from pl_bolts.datasets import DummyDetectionDataset  
>>> from torch.utils.data import DataLoader  
>>> ds = DummyDetectionDataset()  
>>> dl = DataLoader(ds, batch_size=7)
```

Parameters

- ***shapes** – list of shapes
- **num_samples** (`int`) – how many samples to use in this dataset

12.1.3 RandomDataset

```
class pl_bolts.datasets.dummy_dataset.RandomDataset (size, num_samples=250)
```

Bases: `torch.utils.data.Dataset`

Generate a dummy dataset

Example

```
>>> from pl_bolts.datasets import RandomDataset  
>>> from torch.utils.data import DataLoader  
>>> ds = RandomDataset(10)  
>>> dl = DataLoader(ds, batch_size=7)
```

Parameters

- **size** (`int`) – tuple
- **num_samples** (`int`) – number of samples

12.1.4 RandomDictDataset

```
class pl_bolts.datasets.dummy_dataset.RandomDictDataset (size, num_samples=250)
```

Bases: `torch.utils.data.Dataset`

Generate a dummy dataset with a dict structure

Example

```
>>> from pl_bolts.datasets import RandomDictDataset
>>> from torch.utils.data import DataLoader
>>> ds = RandomDictDataset(10)
>>> dl = DataLoader(ds, batch_size=7)
```

Parameters

- **size** `(int)` – tuple
- **num_samples** `(int)` – number of samples

12.1.5 RandomDictStringDataset

class `pl_bolts.datasets.dummy_dataset.RandomDictStringDataset` (*size*,
num_samples=250)

Bases: `torch.utils.data.Dataset`

Generate a dummy dataset with strings

Example

```
>>> from pl_bolts.datasets import RandomDictStringDataset
>>> from torch.utils.data import DataLoader
>>> ds = RandomDictStringDataset(10)
>>> dl = DataLoader(ds, batch_size=7)
```

Parameters

- **size** `(int)` – tuple
- **num_samples** `(int)` – number of samples

ASYNCHRONOUSLOADER

This dataloader behaves identically to the standard pytorch dataloader, but will transfer data asynchronously to the GPU with training. You can also use it to wrap an existing dataloader.

Example:

```
dataloader = AsynchronousLoader(DataLoader(ds, batch_size=16), device=device)

for b in dataloader:
    ...
```

```
class pl_bolts.datamodules.async_dataloader.AsynchronousLoader(data, device=torch.device,
                                                                q_size=10,
                                                                num_batches=None,
                                                                **kwargs)
```

Bases: `object`

Class for asynchronously loading from CPU memory to device memory with `DataLoader`.

Note that this only works for single GPU training, multiGPU uses PyTorch's `DataParallel` or `DistributedDataParallel` which uses its own code for transferring data across GPUs. This could just break or make things slower with `DataParallel` or `DistributedDataParallel`.

Parameters

- **data** (Union[`DataLoader`, `Dataset`]) – The PyTorch Dataset or DataLoader we're using to load.
- **device** (device) – The PyTorch device we are loading to
- **q_size** (int) – Size of the queue used to store the data loaded to the device
- **num_batches** (Optional[int]) – Number of batches to load. This must be set if the dataloader doesn't have a finite `__len__`. It will also override `DataLoader.__len__` if set and `DataLoader` has a `__len__`. Otherwise it can be left as `None`
- ****kwargs** – Any additional arguments to pass to the dataloader if we're constructing one here

LOSSES

This package lists common losses across research domains (This is a work in progress. If you have any losses you want to contribute, please submit a PR!)

Note: this module is a work in progress

14.1 Your Loss

We're cleaning up many of our losses, but in the meantime, submit a PR to add your loss here!

OBJECT DETECTION

These are common losses used in object detection.

15.1 GloU Loss

`pl_bolts.losses.object_detection.giou_loss(preds, target)`

Calculates the generalized intersection over union loss.

It has been proposed in [Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression](#).

Parameters

- **`preds`** (Tensor) – an Nx4 batch of prediction bounding boxes with representation `[x_min, y_min, x_max, y_max]`
- **`target`** (Tensor) – an Mx4 batch of target bounding boxes with representation `[x_min, y_min, x_max, y_max]`

Example

```
>>> import torch
>>> from pl_bolts.losses.object_detection import giou_loss
>>> preds = torch.tensor([[100, 100, 200, 200]])
>>> target = torch.tensor([[150, 150, 250, 250]])
>>> giou_loss(preds, target)
tensor([[1.0794]])
```

Return type Tensor

Returns GloU loss in an NxM tensor containing the pairwise GloU loss for every element in `preds` and `target`, where N is the number of prediction bounding boxes and M is the number of target bounding boxes

15.2 IoU Loss

`pl_bolts.losses.object_detection.iou_loss(preds, target)`

Calculates the intersection over union loss.

Parameters

- **preds** `(Tensor)` – batch of prediction bounding boxes with representation `[x_min, y_min, x_max, y_max]`
- **target** `(Tensor)` – batch of target bounding boxes with representation `[x_min, y_min, x_max, y_max]`

Example

```
>>> import torch
>>> from pl_bolts.losses.object_detection import iou_loss
>>> preds = torch.tensor([[100, 100, 200, 200]])
>>> target = torch.tensor([[150, 150, 250, 250]])
>>> iou_loss(preds, target)
tensor([[0.8571]])
```

Return type `Tensor`

Returns IoU loss

REINFORCEMENT LEARNING

These are common losses used in RL.

16.1 DQN Loss

`pl_bolts.losses.rl.dqn_loss(batch, net, target_net, gamma=0.99)`

Calculates the mse loss using a mini batch from the replay buffer

Parameters

- **batch** (Tuple[`Tensor`, `Tensor`]) – current mini batch of replay data
- **net** (`Module`) – main training network
- **target_net** (`Module`) – target network of the main training network
- **gamma** (`float`) – discount factor

Return type `Tensor`

Returns loss

16.2 Double DQN Loss

`pl_bolts.losses.rl.double_dqn_loss(batch, net, target_net, gamma=0.99)`

Calculates the mse loss using a mini batch from the replay buffer. This uses an improvement to the original DQN loss by using the double dqn. This is shown by using the actions of the train network to pick the value from the target network. This code is heavily commented in order to explain the process clearly

Parameters

- **batch** (Tuple[`Tensor`, `Tensor`]) – current mini batch of replay data
- **net** (`Module`) – main training network
- **target_net** (`Module`) – target network of the main training network
- **gamma** (`float`) – discount factor

Return type `Tensor`

Returns loss

16.3 Per DQN Loss

`pl_bolts.losses.rl.per_dqn_loss` (*batch, batch_weights, net, target_net, gamma=0.99*)

Calculates the mse loss with the priority weights of the batch from the PER buffer

Parameters

- **batch** `(Tuple[Tensor, Tensor])` – current mini batch of replay data
- **batch_weights** `(List)` – how each of these samples are weighted in terms of priority
- **net** `(Module)` – main training network
- **target_net** `(Module)` – target network of the main training network
- **gamma** `(float)` – discount factor

Return type `Tuple[Tensor, ndarray]`

Returns loss and batch_weights

HOW TO USE MODELS

Models are meant to be “bolted” onto your research or production cases.

Bolts are meant to be used in the following ways

17.1 Predicting on your data

Most bolts have pretrained weights trained on various datasets or algorithms. This is useful when you don’t have enough data, time or money to do your own training.

For example, you could use a pretrained VAE to generate features for an image dataset.

```
from pl_bolts.models.self_supervised import SimCLR

weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/simclr/bolts_
↳simclr_imagenet/simclr_imagenet.ckpt'
simclr = SimCLR.load_from_checkpoint(weight_path, strict=False)
encoder = simclr.encoder
encoder.eval()
```

```
for (x, y) in own_data:
    features = encoder(x)
```

The advantage of bolts is that each system can be decomposed and used in interesting ways. For instance, this resnet50 was trained using self-supervised learning (no labels) on Imagenet, and thus might perform better than the same resnet50 trained with labels

```
# trained without labels
from pl_bolts.models.self_supervised import SimCLR

weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/simclr/bolts_
↳simclr_imagenet/simclr_imagenet.ckpt'
simclr = SimCLR.load_from_checkpoint(weight_path, strict=False)
resnet50_unsupervised = simclr.encoder.eval()

# trained with labels
from torchvision.models import resnet50
resnet50_supervised = resnet50(pretrained=True)
```

```
# perhaps the features when trained without labels are much better for classification_
↳ or other tasks
x = image_sample()
unsup_feats = resnet50_unsupervised(x)
sup_feats = resnet50_supervised(x)

# which one will be better?
```

Bolts are often trained on more than just one dataset.

```
from pl_bolts.models.self_supervised import SimCLR

# imagenet weights
weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/simclr/bolts_
↳ simclr_imagenet/simclr_imagenet.ckpt'
simclr = SimCLR.load_from_checkpoint(weight_path, strict=False)

simclr.freeze()
```

17.2 Finetuning on your data

If you have a little bit of data and can pay for a bit of training, it's often better to finetune on your own data.

To finetune you have two options unfrozen finetuning or unfrozen later.

17.2.1 Unfrozen Finetuning

In this approach, we load the pretrained model and unfreeze from the beginning

```
from pl_bolts.models.self_supervised import SimCLR

weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/simclr/bolts_
↳ simclr_imagenet/simclr_imagenet.ckpt'
simclr = SimCLR.load_from_checkpoint(weight_path, strict=False)
resnet50 = simclr.encoder
# don't call .freeze()
```

```
classifier = LogisticRegression(...)

for (x, y) in own_data:
    feats = resnet50(x)
    y_hat = classifier(feats)
    ...
```

Or as a LightningModule

```
class FineTuner(pl.LightningModule):

    def __init__(self, encoder):
        self.encoder = encoder
        self.classifier = LogisticRegression(...)
```

(continues on next page)

(continued from previous page)

```

def training_step(self, batch, batch_idx):
    (x, y) = batch
    feats = self.encoder(x)
    y_hat = self.classifier(feats)
    loss = cross_entropy_with_logits(y_hat, y)
    return loss

trainer = Trainer(gpus=2)
model = FineTuner(resnet50)
trainer.fit(model)

```

Sometimes this works well, but more often it's better to keep the encoder frozen for a while

17.2.2 Freeze then unfreeze

The approach that works best most often is to freeze first then unfreeze later

```

# freeze!
from pl_bolts.models.self_supervised import SimCLR

weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/simclr/bolts_
↳simclr_imagenet/simclr_imagenet.ckpt'
simclr = SimCLR.load_from_checkpoint(weight_path, strict=False)
resnet50 = simclr.encoder
resnet50.eval()

```

```

classifier = LogisticRegression(...)

for epoch in epochs:
    for (x, y) in own_data:
        feats = resnet50(x)
        y_hat = classifier(feats)
        loss = cross_entropy_with_logits(y_hat, y)

    # unfreeze after 10 epochs
    if epoch == 10:
        resnet50.unfreeze()

```

Note: In practice, unfreezing later works MUCH better.

Or in Lightning as a Callback so you don't pollute your research code.

```

class UnFreezeCallback(Callback):

    def on_epoch_end(self, trainer, pl_module):
        if trainer.current_epoch == 10:
            encoder.unfreeze()

trainer = Trainer(gpus=2, callbacks=[UnFreezeCallback()])
model = FineTuner(resnet50)
trainer.fit(model)

```

Unless you still need to mix it into your research code.

```
class FineTuner(pl.LightningModule):

    def __init__(self, encoder):
        self.encoder = encoder
        self.classifier = LogisticRegression(...)

    def training_step(self, batch, batch_idx):

        # option 1 - (not recommended because it's messy)
        if self.trainer.current_epoch == 10:
            self.encoder.unfreeze()

        (x, y) = batch
        feats = self.encoder(x)
        y_hat = self.classifier(feats)
        loss = cross_entropy_with_logits(y_hat, y)
        return loss

    def on_epoch_end(self, trainer, pl_module):
        # a hook is cleaner (but a callback is much better)
        if self.trainer.current_epoch == 10:
            self.encoder.unfreeze()
```

17.2.3 Hyperparameter search

For finetuning to work well, you should try many versions of the model hyperparameters. Otherwise you're unlikely to get the most value out of your data.

```
from pl_bolts.models.autoencoders import VAE

learning_rates = [0.01, 0.001, 0.0001]
hidden_dim = [128, 256, 512]

for lr in learning_rates:
    for hd in hidden_dim:
        vae = VAE(input_height=32, hidden_dim=hd, learning_rate=lr)
        trainer = Trainer()
        trainer.fit(vae)
```

17.3 Train from scratch

If you do have enough data and compute resources, then you could try training from scratch.

```
# get data
train_data = DataLoader(YourDataset)
val_data = DataLoader(YourDataset)

# use any bolts model without pretraining
model = VAE()

# fit!
```

(continues on next page)

(continued from previous page)

```
trainer = Trainer(gpus=2)
trainer.fit(model, train_dataloader=train_data, val_data_loaders=val_data)
```

Note: For this to work well, make sure you have enough data and time to train these models!

17.4 For research

What separates bolts from all the other libraries out there is that bolts is built by and used by AI researchers. This means every single bolt is modularized so that it can be easily extended or mixed with arbitrary parts of the rest of the code-base.

17.4.1 Extending work

Perhaps a research project requires modifying a part of a known approach. In this case, you're better off only changing that part of a system that is already known to perform well. Otherwise, you risk not implementing the work correctly.

Example 1: Changing the prior or approx posterior of a VAE

```
from pl_bolts.models.autoencoders import VAE

class MyVAEFlavor(VAE):

    def init_prior(self, z_mu, z_std):
        P = MyPriorDistribution

        # default is standard normal
        # P = distributions.normal.Normal(loc=torch.zeros_like(z_mu), scale=torch.
        ↪ ones_like(z_std))
        return P

    def init_posterior(self, z_mu, z_std):
        Q = MyPosteriorDistribution
        # default is normal(z_mu, z_sigma)
        # Q = distributions.normal.Normal(loc=z_mu, scale=z_std)
        return Q
```

And of course train it with lightning.

```
model = MyVAEFlavor()
trainer = Trainer()
trainer.fit(model)
```

In just a few lines of code you changed something fundamental about a VAE... This means you can iterate through ideas much faster knowing that the bolt implementation and the training loop are CORRECT and TESTED.

If your model doesn't work with the new P, Q, then you can discard that research idea much faster than trying to figure out if your VAE implementation was correct, or if your training loop was correct.

Example 2: Changing the generator step of a GAN

```
from pl_bolts.models.gans import GAN

class FancyGAN(GAN):

    def generator_step(self, x):
        # sample noise
        z = torch.randn(x.shape[0], self.hparams.latent_dim)
        z = z.type_as(x)

        # generate images
        self.generated_imgs = self(z)

        # ground truth result (ie: all real)
        real = torch.ones(x.size(0), 1)
        real = real.type_as(x)
        g_loss = self.generator_loss(real)

        tqdm_dict = {'g_loss': g_loss}
        output = OrderedDict({
            'loss': g_loss,
            'progress_bar': tqdm_dict,
            'log': tqdm_dict
        })
        return output
```

Example 3: Changing the way the loss is calculated in a contrastive self-supervised learning approach

```
from pl_bolts.models.self_supervised import AMDIM

class MyDIM(AMDIM):

    def validation_step(self, batch, batch_nb):
        [img_1, img_2], labels = batch

        # generate features
        r1_x1, r5_x1, r7_x1, r1_x2, r5_x2, r7_x2 = self.forward(img_1, img_2)

        # Contrastive task
        loss, lgt_reg = self.contrastive_task((r1_x1, r5_x1, r7_x1), (r1_x2, r5_x2,
↪r7_x2))
        unsupervised_loss = loss.sum() + lgt_reg

        result = {
            'val_nce': unsupervised_loss
        }
        return result
```

17.4.2 Importing parts

All the bolts are modular. This means you can also arbitrarily mix and match fundamental blocks from across approaches.

Example 1: Use the VAE encoder for a GAN as a generator

```
from pl_bolts.models.gans import GAN
from pl_bolts.models.autoencoders.basic_vae import Encoder

class FancyGAN(GAN):

    def init_generator(self, img_dim):
        generator = Encoder(...)
        return generator

trainer = Trainer(...)
trainer.fit(FancyGAN())
```

Example 2: Use the contrastive task of AMDIM in CPC

```
from pl_bolts.models.self_supervised import AMDIM, CPCV2

default_amdim_task = AMDIM().contrastive_task
model = CPCV2(contrastive_task=default_amdim_task, encoder='cpc_default')
# you might need to modify the cpc encoder depending on what you use
```

17.4.3 Compose new ideas

You may also be interested in creating completely new approaches that mix and match all sorts of different pieces together

```
# this model is for illustration purposes, it makes no research sense but it's
↳ intended to show
# that you can be as creative and expressive as you want.
class MyNewContrastiveApproach(pl.LightningModule):

    def __init__(self):
        suouer().__init__()

        self.gan = GAN()
        self.vae = VAE()
        self.amdim = AMDIM()
        self.cpc = CPCV2

    def training_step(self, batch, batch_idx):
        (x, y) = batch

        feat_a = self.gan.generator(x)
        feat_b = self.vae.encoder(x)

        unsup_loss = self.amdim(feat_a) + self.cpc(feat_b)

        vae_loss = self.vae._step(batch)
        gan_loss = self.gan.generator_loss(x)
```

(continues on next page)

(continued from previous page)

```
return unsup_loss + vae_loss + gan_loss
```

CLASSIC ML MODELS

This module implements classic machine learning models in PyTorch Lightning, including linear regression and logistic regression. Unlike other libraries that implement these models, here we use PyTorch to enable multi-GPU, multi-TPU and half-precision training.

18.1 Linear Regression

Linear regression fits a linear model between a real-valued target variable y and one or more features X . We estimate the regression coefficients that minimize the mean squared error between the predicted and true target values.

We formulate the linear regression model as a single-layer neural network. By default we include only one neuron in the output layer, although you can specify the *output_dim* yourself.

Add either L1 or L2 regularization, or both, by specifying the regularization strength (default 0).

```
from pl_bolts.models.regression import LinearRegression
import pytorch_lightning as pl
from pl_bolts.datamodules import SklearnDataModule
from sklearn.datasets import load_boston
```

```
X, y = load_boston(return_X_y=True)
loaders = SklearnDataModule(X, y)

model = LinearRegression(input_dim=13)
trainer = pl.Trainer()
trainer.fit(model, train_dataloader=loaders.train_dataloader(), val_
↳ dataloaders=loaders.val_dataloader())
trainer.test(test_dataloaders=loaders.test_dataloader())
```

```
class pl_bolts.models.regression.linear_regression.LinearRegression(input_dim,
                                                                    out-
                                                                    put_dim=1,
                                                                    bias=True,
                                                                    learn-
                                                                    ing_rate=0.0001,
                                                                    opti-
                                                                    mizer=torch.optim.Adam,
                                                                    l1_strength=0.0,
                                                                    l2_strength=0.0,
                                                                    **kwargs)
```

Bases: pytorch_lightning.LightningModule

Linear regression model implementing - with optional L1/L2 regularization $\min_{\{W\}} \| (Wx + b) - y \|_2^2$

Parameters

- `input_dim` (int) – number of dimensions of the input (1+)
 - `output_dim` (int) – number of dimensions of the output (default=1)
 - `bias` (bool) – If false, will not use $+b$
 - `learning_rate` (float) – learning_rate for the optimizer
 - `optimizer` (Optimizer) – the optimizer to use (default='Adam')
 - `l1_strength` (float) – L1 regularization strength (default=None)
 - `l2_strength` (float) – L2 regularization strength (default=None)
-

18.2 Logistic Regression

Logistic regression is a linear model used for classification, i.e. when we have a categorical target variable. This implementation supports both binary and multi-class classification.

In the binary case, we formulate the logistic regression model as a one-layer neural network with one neuron in the output layer and a sigmoid activation function. In the multi-class case, we use a single-layer neural network but now with k neurons in the output, where k is the number of classes. This is also referred to as multinomial logistic regression.

Add either L1 or L2 regularization, or both, by specifying the regularization strength (default 0).

```
from sklearn.datasets import load_iris
from pl_bolts.models.regression import LogisticRegression
from pl_bolts.datamodules import SklearnDataModule
import pytorch_lightning as pl

# use any numpy or sklearn dataset
X, y = load_iris(return_X_y=True)
dm = SklearnDataModule(X, y)

# build model
model = LogisticRegression(input_dim=4, num_classes=3)

# fit
trainer = pl.Trainer(tpu_cores=8, precision=16)
trainer.fit(model, train_dataloader=dm.train_dataloader(), val_dataloaders=dm.val_
    ↪dataloader())

trainer.test(test_dataloaders=dm.test_dataloader(batch_size=12))
```

Any input will be flattened across all dimensions except the first one (batch). This means images, sound, etc... work out of the box.

```
# create dataset
dm = MNISTDataModule(num_workers=0, data_dir=tmpdir)

model = LogisticRegression(input_dim=28 * 28, num_classes=10, learning_rate=0.001)
```

(continues on next page)

(continued from previous page)

```

model.prepare_data = dm.prepare_data
model.train_dataloader = dm.train_dataloader
model.val_dataloader = dm.val_dataloader
model.test_dataloader = dm.test_dataloader

trainer = pl.Trainer(max_epochs=2)
trainer.fit(model)
trainer.test(model)
# {test_acc: 0.92}

```

```

class pl_bolts.models.regression.logistic_regression.LogisticRegression(input_dim,
                                                                    num_classes,
                                                                    bias=True,
                                                                    learning_rate=0.0001,
                                                                    optimizer=torch.optim.Adam,
                                                                    l1_strength=0.0,
                                                                    l2_strength=0.0,
                                                                    **kwargs)

```

Bases: `pytorch_lightning.LightningModule`

Logistic regression model

Parameters

- **input_dim** (int) – number of dimensions of the input (at least 1)
- **num_classes** (int) – number of class labels (binary: 2, multi-class: >2)
- **bias** (bool) – specifies if a constant or intercept should be fitted (equivalent to `fit_intercept` in `sklearn`)
- **learning_rate** (float) – learning_rate for the optimizer
- **optimizer** (Optimizer) – the optimizer to use (default='Adam')
- **l1_strength** (float) – L1 regularization strength (default=None)
- **l2_strength** (float) – L2 regularization strength (default=None)

AUTOENCODERS

This section houses autoencoders and variational autoencoders.

19.1 Basic AE

This is the simplest autoencoder. You can use it like so

```
from pl_bolts.models.autoencoders import AE

model = AE()
trainer = Trainer()
trainer.fit(model)
```

You can override any part of this AE to build your own variation.

```
from pl_bolts.models.autoencoders import AE

class MyAEFlavor(AE):

    def init_encoder(self, hidden_dim, latent_dim, input_width, input_height):
        encoder = YourSuperFancyEncoder(...)
        return encoder
```

You can use the pretrained models present in bolts.

CIFAR-10 pretrained model:

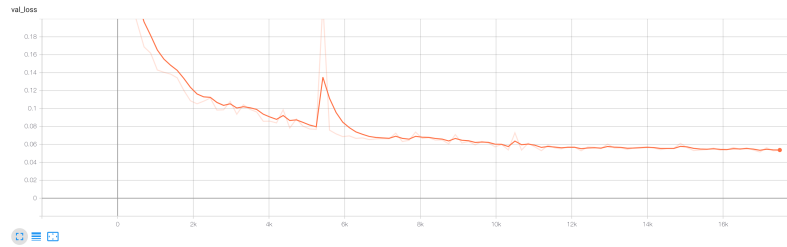
```
from pl_bolts.models.autoencoders import AE

ae = AE(input_height=32)
print(AE.pretrained_weights_available())
ae = ae.from_pretrained('cifar10-resnet18')

ae.freeze()
```

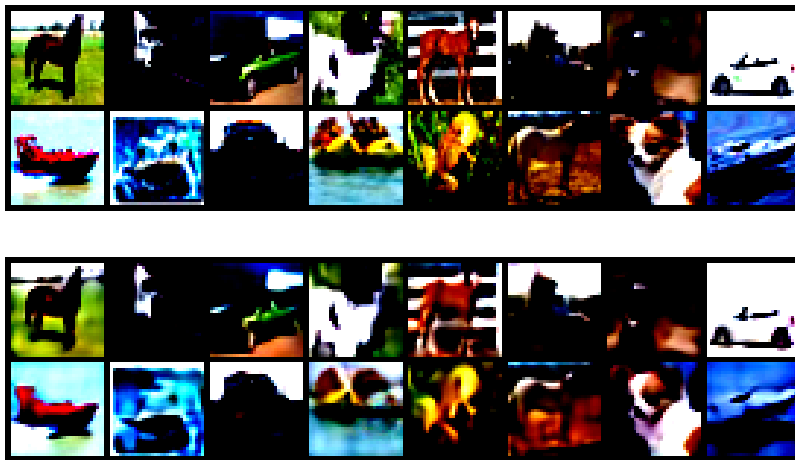
- [Tensorboard for AE on CIFAR-10](#)

Training:



Reconstructions:

Both input and generated images are normalized versions as the training was done with such images.



```
class pl_bolts.models.autoencoders.AE(input_height, enc_type='resnet18', first_conv=False,
                                       maxpool1=False, enc_out_dim=512, latent_dim=256, lr=0.0001, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Standard AE

Model is available pretrained on different datasets:

Example:

```
# not pretrained
ae = AE()

# pretrained on cifar10
ae = AE.from_pretrained('cifar10-resnet18')
```

Parameters

- **input_height** `(int)` – height of the images
- **enc_type** `(str)` – option between `resnet18` or `resnet50`

- `first_conv` (bool) – use standard kernel_size 7, stride 2 at start or replace it with kernel_size 3, stride 1 conv
- `maxpool1` (bool) – use standard maxpool to reduce spatial dim of feat by a factor of 2
- `enc_out_dim` (int) – set according to the out_channel count of encoder used (512 for resnet18, 2048 for resnet50)
- `latent_dim` (int) – dim of latent space
- `lr` (float) – learning rate for Adam

19.1.1 Variational Autoencoders

19.2 Basic VAE

Use the VAE like so.

```
from pl_bolts.models.autoencoders import VAE

model = VAE()
trainer = Trainer()
trainer.fit(model)
```

You can override any part of this VAE to build your own variation.

```
from pl_bolts.models.autoencoders import VAE

class MyVAEFlavor(VAE):

    def get_posterior(self, mu, std):
        # do something other than the default
        # P = self.get_distribution(self.prior, loc=torch.zeros_like(mu), scale=torch.
        ↪ ones_like(std))

        return P
```

You can use the pretrained models present in bolts.

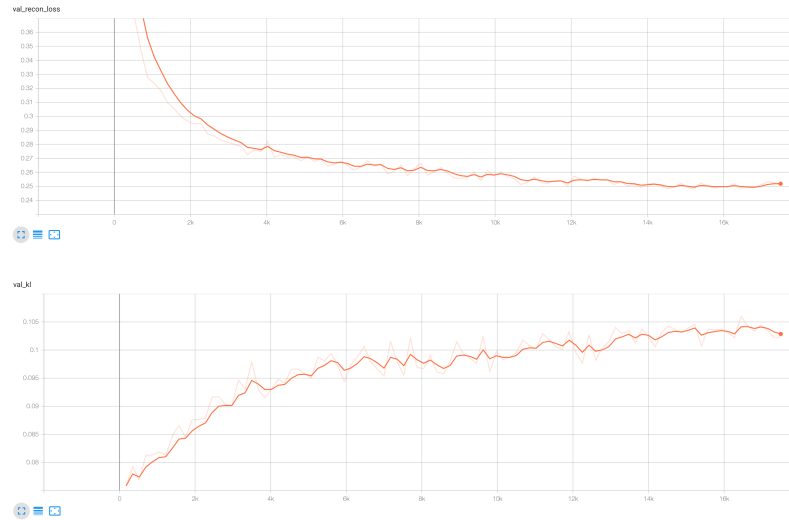
CIFAR-10 pretrained model:

```
from pl_bolts.models.autoencoders import VAE

vae = VAE(input_height=32)
print(VAE.pretrained_weights_available())
vae = vae.from_pretrained('cifar10-resnet18')

vae.freeze()
```

- Tensorboard for VAE on CIFAR-10



Training:

Reconstructions:

Both input and generated images are normalized versions as the training was done with such images.



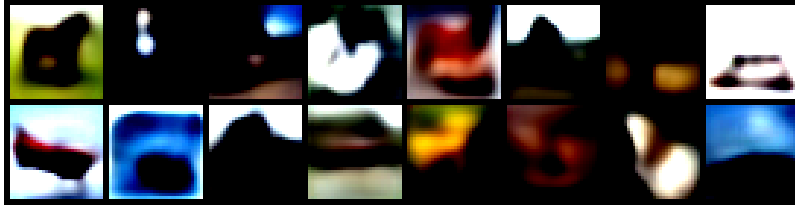
STL-10 pretrained model:

```
from pl_bolts.models.autoencoders import VAE

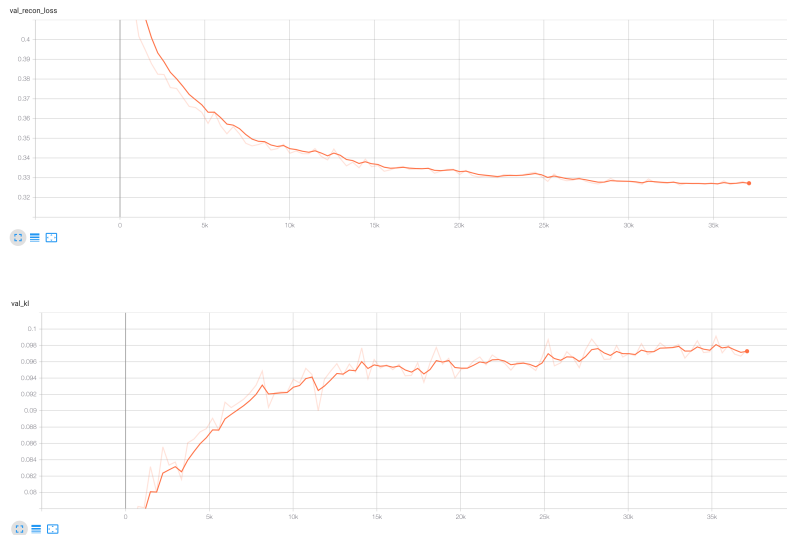
vae = VAE(input_height=96, first_conv=True)
print(VAE.pretrained_weights_available())
vae = vae.from_pretrained('cifar10-resnet18')

vae.freeze()
```

- Tensorboard for VAE on STL-10



Training:



```
class pl_bolts.models.autoencoders.VAE (input_height, enc_type='resnet18', first_conv=False,
                                         maxpool1=False, enc_out_dim=512, kl_coeff=0.1,
                                         latent_dim=256, lr=0.0001, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Standard VAE with Gaussian Prior and approx posterior.

Model is available pretrained on different datasets:

Example:

```
# not pretrained
vae = VAE()

# pretrained on cifar10
vae = VAE.from_pretrained('cifar10-resnet18')

# pretrained on stl10
vae = VAE.from_pretrained('stl10-resnet18')
```

Parameters

- **input_height** (`int`) – height of the images
- **enc_type** (`str`) – option between `resnet18` or `resnet50`

- **first_conv** (bool) – use standard kernel_size 7, stride 2 at start or replace it with kernel_size 3, stride 1 conv
- **maxpool1** (bool) – use standard maxpool to reduce spatial dim of feat by a factor of 2
- **enc_out_dim** (int) – set according to the out_channel count of encoder used (512 for resnet18, 2048 for resnet50)
- **kl_coeff** (float) – coefficient for kl term of the loss
- **latent_dim** (int) – dim of latent space
- **lr** (float) – learning rate for Adam

CONVOLUTIONAL ARCHITECTURES

This package lists contributed convolutional architectures.

20.1 GPT-2

class `pl_bolts.models.vision.image_gpt.gpt2.GPT2` (*embed_dim*, *heads*, *layers*,
num_positions, *vocab_size*,
num_classes)

Bases: `pytorch_lightning.LightningModule`

GPT-2 from [language Models are Unsupervised Multitask Learners](#)

Paper by: Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever

Implementation contributed by:

- [Teddy Koker](#)

Example:

```
from pl_bolts.models import GPT2

seq_len = 17
batch_size = 32
vocab_size = 16
x = torch.randint(0, vocab_size, (seq_len, batch_size))
model = GPT2(embed_dim=32, heads=2, layers=2, num_positions=seq_len, vocab_
↪size=vocab_size, num_classes=4)
results = model(x)
```

forward (*x*, *classify=False*)

Expect input as shape [sequence len, batch] If classify, return classification logits

20.2 Image GPT

```
class pl_bolts.models.vision.image_gpt.igpt_module.ImageGPT (embed_dim=16,  
                                                         heads=2,         lay-  
                                                         ers=2,      pixels=28,  
                                                         vocab_size=16,  
                                                         num_classes=10,  
                                                         classify=False,  
                                                         batch_size=64,  
                                                         learning_rate=0.01,  
                                                         steps=25000,  
                                                         data_dir='.',  
                                                         num_workers=8,  
                                                         **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Paper: [Generative Pretraining from Pixels](#) [original paper [code](#)].

Paper by: Mark Che, Alec Radford, Rewon Child, Jeff Wu, Heewoo Jun, Prafulla Dhariwal, David Luan, Ilya Sutskever

Implementation contributed by:

- [Teddy Koker](#)

Original repo with results and more implementation details:

- <https://github.com/teddykoker/image-gpt>

Example Results (Photo credits: Teddy Koker):



Default arguments:

Table 1: Argument Defaults

Argument	Default	iGPT-S (Chen et al.)
<code>-embed_dim</code>	16	512
<code>-heads</code>	2	8
<code>-layers</code>	8	24
<code>-pixels</code>	28	32
<code>-vocab_size</code>	16	512
<code>-num_classes</code>	10	10
<code>-batch_size</code>	64	128
<code>-learning_rate</code>	0.01	0.01
<code>-steps</code>	25000	1000000

Example:

```
import pytorch_lightning as pl
from pl_bolts.models.vision import ImageGPT

dm = MNISTDataModule('.')
model = ImageGPT(dm)

pl.Trainer(gpu=4).fit(model)
```

As script:

```
cd pl_bolts/models/vision/image_gpt
python igpt_module.py --learning_rate 1e-2 --batch_size 32 --gpus 4
```

Parameters

- `embed_dim` (int) – the embedding dim
- `heads` (int) – number of attention heads
- `layers` (int) – number of layers
- `pixels` (int) – number of input pixels
- `vocab_size` (int) – vocab size
- `num_classes` (int) – number of classes in the input
- `classify` (bool) – true if should classify
- `batch_size` (int) – the batch size
- `learning_rate` (float) – learning rate
- `steps` (int) – number of steps for cosine annealing
- `data_dir` (str) – where to store data
- `num_workers` (int) – num_data workers

20.3 Pixel CNN

class `pl_bolts.models.vision.pixel_cnn.PixelCNN` (*input_channels*, *hidden_channels=256*, *num_blocks=5*)

Bases: `torch.nn.Module`

Implementation of Pixel CNN.

Paper authors: Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, Koray Kavukcuoglu

Implemented by:

- William Falcon

Example:

```
>>> from pl_bolts.models.vision import PixelCNN
>>> import torch
...
>>> model = PixelCNN(input_channels=3)
>>> x = torch.rand(5, 3, 64, 64)
>>> out = model(x)
...
>>> out.shape
torch.Size([5, 3, 64, 64])
```

20.4 UNet

class `pl_bolts.models.vision.unet.UNet` (*num_classes*, *input_channels=3*, *num_layers=5*, *features_start=64*, *bilinear=False*)

Bases: `torch.nn.Module`

Paper: [U-Net: Convolutional Networks for Biomedical Image Segmentation](#)

Paper authors: Olaf Ronneberger, Philipp Fischer, Thomas Brox

Implemented by:

- Annika Brundyn
- Akshay Kulkarni

Parameters

- **num_classes** `(int)` – Number of output classes required
- **input_channels** `(int)` – Number of channels in input images (default 3)
- **num_layers** `(int)` – Number of layers in each side of U-net (default 5)
- **features_start** `(int)` – Number of features in first layer (default 64)
- **bilinear** `(bool)` – Whether to use bilinear interpolation or transposed convolutions (default) for upsampling.

20.5 Semantic Segmentation

Model template to use for semantic segmentation tasks. The model uses a UNet architecture by default. Override any part of this model to build your own variation.

```
from pl_bolts.models.vision import SemSegment
from pl_bolts.datamodules import KittiDataModule
import pytorch_lightning as pl

dm = KittiDataModule('path/to/kitt/dataset/', batch_size=4)
model = SemSegment(datamodule=dm)
trainer = pl.Trainer()
trainer.fit(model)
```

```
class pl_bolts.models.vision.segmentation.SemSegment (lr=0.01, num_classes=19,
                                                    num_layers=5, features_start=64,
                                                    bilinear=False)
```

Bases: `pytorch_lightning.LightningModule`

Basic model for semantic segmentation. Uses UNet architecture by default.

The default parameters in this model are for the KITTI dataset. Note, if you'd like to use this model as is, you will first need to download the KITTI dataset yourself. You can download the dataset [here](#).

Implemented by:

- [Annika Brundyn](#)

Parameters

- **num_layers** `(int)` – number of layers in each side of U-net (default 5)
- **features_start** `(int)` – number of features in first layer (default 64)
- **bilinear** `(bool)` – whether to use bilinear interpolation (True) or transposed convolutions (default) for upsampling.
- **lr** `(float)` – learning (default 0.01)

GANs

Collection of Generative Adversarial Networks

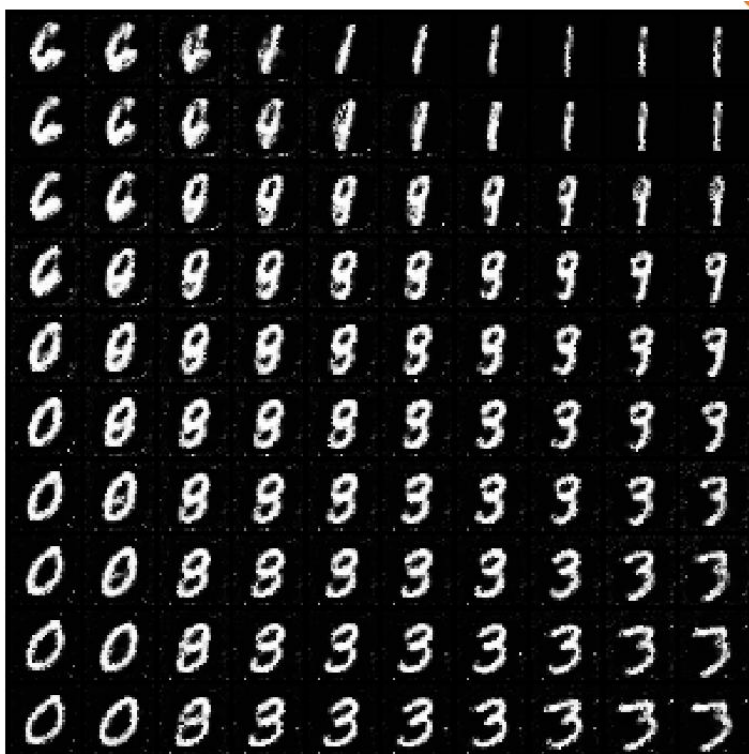
21.1 Basic GAN

This is a vanilla GAN. This model can work on any dataset size but results are shown for MNIST. Replace the encoder, decoder or any part of the training loop to build a new method, or simply finetune on your data.

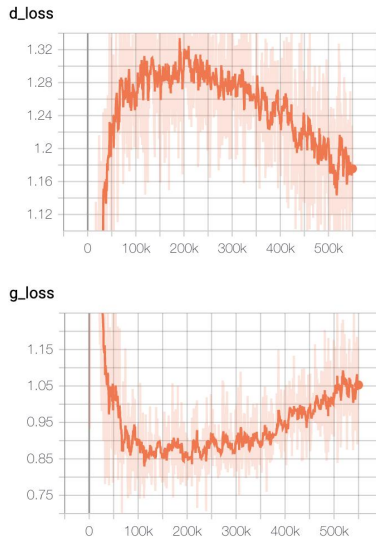
Implemented by:

- William Falcon

Example outputs:



Loss curves:



```
from pl_bolts.models.gans import GAN
...
gan = GAN()
trainer = Trainer()
trainer.fit(gan)
```

class `pl_bolts.models.gans.GAN`(*input_channels*, *input_height*, *input_width*, *latent_dim*=32, *learning_rate*=0.0002, ***kwargs*)

Bases: `pytorch_lightning.LightningModule`

Vanilla GAN implementation.

Example:

```
from pl_bolts.models.gan import GAN

m = GAN()
Trainer(gpus=2).fit(m)
```

Example CLI:

```
# mnist
python basic_gan_module.py --gpus 1

# imagenet
python basic_gan_module.py --gpus 1 --dataset 'imagenet2012'
--data_dir /path/to/imagenet/folder/ --meta_dir ~/path/to/meta/bin/folder
--batch_size 256 --learning_rate 0.0001
```

Parameters

- **input_channels** (int) – number of channels of an image
- **input_height** (int) – image height
- **input_width** (int) – image width
- **latent_dim** (int) – emb dim for encoder
- **learning_rate** (float) – the learning rate

forward(z)

Generates an image given input noise z

Example:

```
z = torch.rand(batch_size, latent_dim)
gan = GAN.load_from_checkpoint(PATH)
img = gan(z)
```

21.2 DCGAN

DCGAN implementation from the paper [Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks](#). The implementation is based on the version from PyTorch's [examples](#).

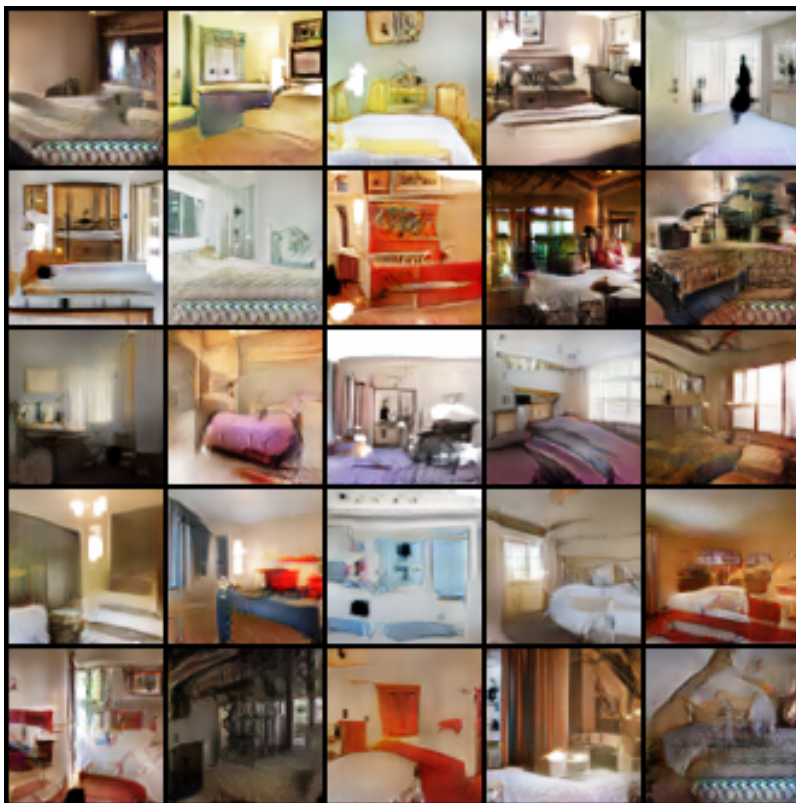
Implemented by:

- [Christoph Clement](#)

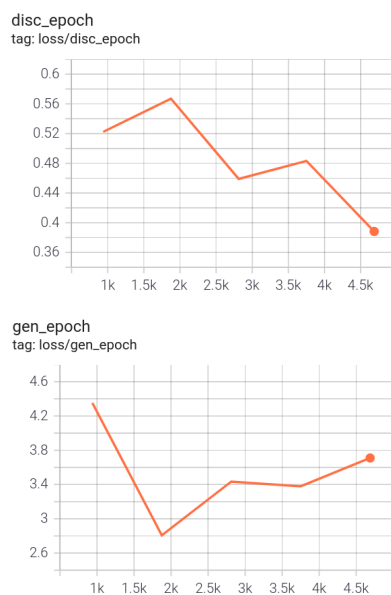
Example MNIST outputs:



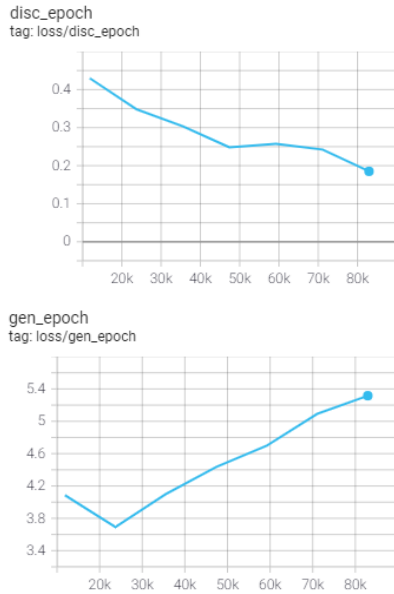
Example LSUN bedroom outputs:



MNIST Loss curves:



LSUN Loss curves:



```
class pl_bolts.models.gans.DCGAN (beta1=0.5, feature_maps_gen=64, feature_maps_disc=64,
                                  image_channels=1, latent_dim=100, learning_rate=0.0002,
                                  **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

DCGAN implementation.

Example:

```
from pl_bolts.models.gan import DCGAN

m = DCGAN()
Trainer(gpus=2).fit(m)
```

Example CLI:

```
# mnist
python dcgan_module.py --gpus 1

# cifar10
python dcgan_module.py --gpus 1 --dataset cifar10 --image_channels 3
```

Parameters

- **beta1** `(float)` – Beta1 value for Adam optimizer
- **feature_maps_gen** `(int)` – Number of feature maps to use for the generator
- **feature_maps_disc** `(int)` – Number of feature maps to use for the discriminator
- **image_channels** `(int)` – Number of channels of the images from the dataset
- **latent_dim** `(int)` – Dimension of the latent space
- **learning_rate** `(float)` – Learning rate

forward (*noise*)

Generates an image given input noise

Example:

```
noise = torch.rand(batch_size, latent_dim)
gan = GAN.load_from_checkpoint(PATH)
img = gan(noise)
```

Return type `Tensor`

REINFORCEMENT LEARNING

This module is a collection of common RL approaches implemented in Lightning.

22.1 Module authors

Contributions by: [Donal Byrne](#)

- DQN
 - Double DQN
 - Dueling DQN
 - Noisy DQN
 - NStep DQN
 - Prioritized Experience Replay DQN
 - Reinforce
 - Vanilla Policy Gradient
-

Note: RL models currently only support CPU and single GPU training with *distributed_backend=dp*. Full GPU support will be added in later updates.

22.2 DQN Models

The following models are based on DQN. DQN uses value based learning where it is deciding what action to take based on the model's current learned value (V), or the state action value (Q) of the current state. These values are defined as the discounted total reward of the agents state or state action pair.

22.2.1 Deep-Q-Network (DQN)

DQN model introduced in [Playing Atari with Deep Reinforcement Learning](#). Paper authors: Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller.

Original implementation by: [Donal Byrne](#)

The DQN was introduced in [Playing Atari with Deep Reinforcement Learning](#) by researchers at DeepMind. This took the concept of tabular Q learning and scaled it to much larger problems by approximating the Q function using a deep neural network.

The goal behind DQN was to take the simple control method of Q learning and scale it up in order to solve complicated tasks. As well as this, the method needed to be stable. The DQN solves these issues with the following additions.

Approximated Q Function

Storing Q values in a table works well in theory, but is completely unscalable. Instead, the authors approximate the Q function using a deep neural network. This allows the DQN to be used for much more complicated tasks

Replay Buffer

Similar to supervised learning, the DQN learns on randomly sampled batches of previous data stored in an Experience Replay Buffer. The ‘target’ is calculated using the Bellman equation

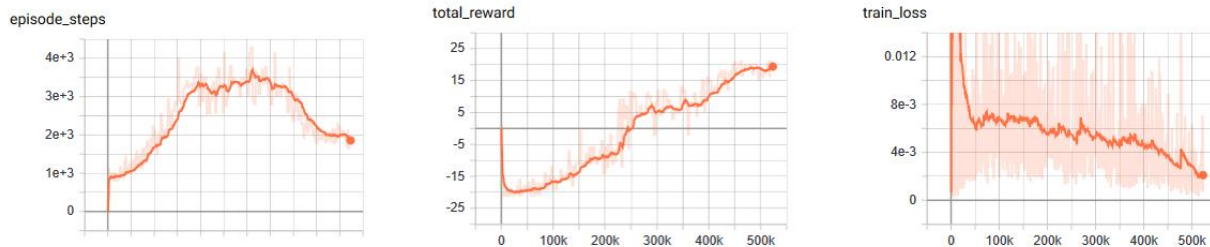
$$Q(s, a) < -(r + \gamma \max_{a' \in A} Q(s', a'))^2$$

and then we optimize using SGD just like a standard supervised learning problem.

$$L = (Q(s, a) - (r + \gamma \max_{a' \in A} Q(s', a')))^2$$

DQN Results

DQN: Pong



Example:

```
from pl_bolts.models.rl import DQN
dqn = DQN("PongNoFrameskip-v4")
trainer = Trainer()
trainer.fit(dqn)
```

```
class pl_bolts.models.rl.dqn_model.DQN(env,
                                         eps_start=1.0,      eps_end=0.02,
                                         eps_last_frame=150000, sync_rate=1000,
                                         gamma=0.99,          learning_rate=0.0001,
                                         batch_size=32,         replay_size=100000,
                                         warm_start_size=10000,  avg_reward_len=100,
                                         min_episode_reward=-21, seed=123,
                                         batches_per_epoch=1000, n_steps=1, **kwargs)

Bases: pytorch_lightning.LightningModule
```

Basic DQN Model

PyTorch Lightning implementation of [DQN](#) Paper authors: Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.dqn_model import DQN
...
>>> model = DQN("PongNoFrameskip-v4")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: This example is based on: https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter06/02_dqn_pong.py

Note: Currently only supports CPU and single GPU training with *distributed_backend=dp*

Parameters

- **env** (str) – gym environment tag
- **eps_start** (float) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (float) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (int) – the final frame in for the decrease of epsilon. At this frame $\epsilon = \epsilon_{\text{end}}$
- **sync_rate** (int) – the number of iterations between syncing up the target network with the train network
- **gamma** (float) – discount factor
- **learning_rate** (float) – learning rate
- **batch_size** (int) – size of minibatch pulled from the DataLoader
- **replay_size** (int) – total capacity of the replay buffer
- **warm_start_size** (int) – how many random steps through the environment to be carried out at the start of training to fill the buffer with a starting point
- **avg_reward_len** (int) – how many episodes to take into account when calculating the avg reward
- **min_episode_reward** (int) – the minimum score that can be achieved in an episode. Used for filling the avg buffer before training begins
- **seed** (int) – seed value for all RNG used
- **batches_per_epoch** (int) – number of batches per epoch
- **n_steps** (int) – size of n step look ahead

static add_model_specific_args (*arg_parser*)
Adds arguments for DQN model

Note: These params are fine tuned for Pong env.

Parameters *arg_parser* (`ArgumentParser`) – parent parser

Return type `ArgumentParser`

build_networks ()
Initializes the DQN train and target networks

Return type `None`

configure_optimizers ()
Initialize Adam optimizer

Return type `List[Optimizer]`

forward (*x*)
Passes in a state *x* through the network and gets the q_values of each action as an output

Parameters *x* (`Tensor`) – environment state

Return type `Tensor`

Returns q values

static make_environment (*env_name*, *seed=None*)
Initialise gym environment

Parameters

- *env_name* (`str`) – environment name or tag
- *seed* (`Optional[int]`) – value to seed the environment RNG for reproducibility

Return type `object`

Returns gym environment

populate (*warm_start*)
Populates the buffer with initial experience

Return type `None`

run_n_episodes (*env*, *n_episodes=1*, *epsilon=1.0*)
Carries out N episodes of the environment with the current agent

Parameters

- *env* – environment to use, either train environment or test environment
- *n_episodes* (`int`) – number of episodes to run
- *epsilon* (`float`) – epsilon value for DQN agent

Return type `List[int]`

test_dataloader ()
Get test loader

Return type `DataLoader`

test_epoch_end (*outputs*)

Log the avg of the test results

Return type `Dict[str, Tensor]`

test_step (**args, **kwargs*)

Evaluate the agent for 10 episodes

Return type `Dict[str, Tensor]`

train_batch ()

Contains the logic for generating a new batch of data to be passed to the DataLoader

Return type `Tuple[Tensor, Tensor, Tensor, Tensor, Tensor]`

Returns yields a Experience tuple containing the state, action, reward, done and next_state.

train_dataloader ()

Get train loader

Return type `DataLoader`

training_step (*batch, _*)

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch recieved

Parameters

- **batch** `(Tuple[Tensor, Tensor])` – current mini batch of replay data
- **_** – batch number, not used

Return type `OrderedDict`

Returns Training loss and log metrics

22.2.2 Double DQN

Double DQN model introduced in [Deep Reinforcement Learning with Double Q-learning](#) Paper authors: Hado van Hasselt, Arthur Guez, David Silver

Original implementation by: [Donal Byrne](#)

The original DQN tends to overestimate Q values during the Bellman update, leading to instability and is harmful to training. This is due to the max operation in the Bellman equation.

We are constantly taking the max of our agents estimates during our update. This may seem reasonable, if we could trust these estimates. However during the early stages of training, the estimates for these values will be off center and can lead to instability in training until our estimates become more reliable

The Double DQN fixes this overestimation by choosing actions for the next state using the main trained network but uses the values of these actions from the more stable target network. So we are still going to take the greedy action, but the value will be less “optimisitic” because it is chosen by the target network.

DQN expected return

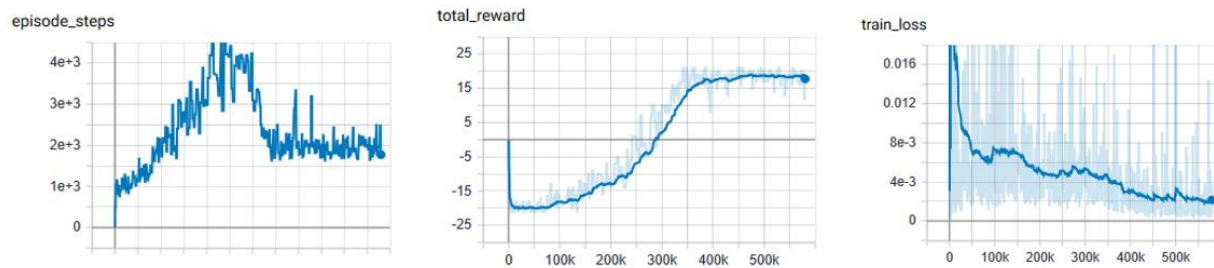
$$Q(s_t, a_t) = r_t + \gamma * \max_{Q'}(S_{t+1}, a)$$

Double DQN expected return

$$Q(s_t, a_t) = r_t + \gamma * \max Q'(S_{t+1}, \arg \max_Q(S_{t+1}, a))$$

Double DQN Results

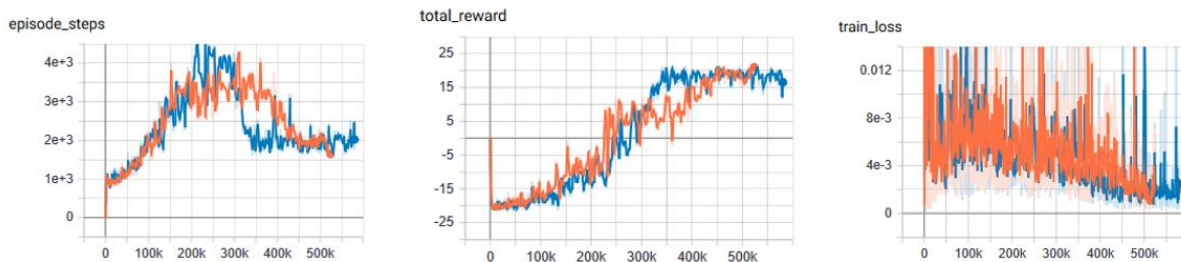
Double DQN: Pong



DQN vs Double DQN: Pong

orange: DQN

blue: Double DQN



Example:

```
from pl_bolts.models.rl import DoubleDQN
ddqn = DoubleDQN("PongNoFrameskip-v4")
trainer = Trainer()
trainer.fit(ddqn)
```

```
class pl_bolts.models.rl.double_dqn_model.DoubleDQN(env,
                                                    eps_start=1.0,
                                                    eps_end=0.02,
                                                    eps_last_frame=150000,
                                                    sync_rate=1000, gamma=0.99,
                                                    learning_rate=0.0001,
                                                    batch_size=32,
                                                    replay_size=100000,
                                                    warm_start_size=10000,
                                                    avg_reward_len=100,
                                                    min_episode_reward=-21,
                                                    seed=123,
                                                    batches_per_epoch=1000,
                                                    n_steps=1, **kwargs)
```

Bases: `pl_bolts.models.rl.dqn_model.DQN`

Double Deep Q-network (DDQN) PyTorch Lightning implementation of [Double DQN](#)

Paper authors: Hado van Hasselt, Arthur Guez, David Silver

Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.double_dqn_model import DoubleDQN
...
>>> model = DoubleDQN("PongNoFrameskip-v4")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: This example is based on https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter08/03_dqn_double.py

Note: Currently only supports CPU and single GPU training with *distributed_backend=dp*

Parameters

- **env** (str) – gym environment tag
- **eps_start** (float) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (float) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (int) – the final frame in for the decrease of epsilon. At this frame $\text{epsilon} = \text{eps_end}$
- **sync_rate** (int) – the number of iterations between syncing up the target network with the train network
- **gamma** (float) – discount factor
- **learning_rate** (float) – learning rate
- **batch_size** (int) – size of minibatch pulled from the DataLoader
- **replay_size** (int) – total capacity of the replay buffer
- **warm_start_size** (int) – how many random steps through the environment to be carried out at the start of training to fill the buffer with a starting point
- **avg_reward_len** (int) – how many episodes to take into account when calculating the avg reward
- **min_episode_reward** (int) – the minimum score that can be achieved in an episode. Used for filling the avg buffer before training begins
- **seed** (int) – seed value for all RNG used
- **batches_per_epoch** (int) – number of batches per epoch
- **n_steps** (int) – size of n step look ahead

training_step (batch, _)

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch recieved

Parameters

- `batch` (Tuple[`Tensor`, `Tensor`]) – current mini batch of replay data
- `_` – batch number, not used

Return type `OrderedDict`**Returns** Training loss and log metrics

22.2.3 Dueling DQN

Dueling DQN model introduced in [Dueling Network Architectures for Deep Reinforcement Learning](#) Paper authors: Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas

Original implementation by: [Donal Byrne](#)

The Q value that we are trying to approximate can be divided into two parts, the value state $V(s)$ and the ‘advantage’ of actions in that state $A(s, a)$. Instead of having one full network estimate the entire Q value, Dueling DQN uses two estimator heads in order to separate the estimation of the two parts.

The value is the same as in value iteration. It is the discounted expected reward achieved from state s . Think of the value as the ‘base reward’ from being in state s .

The advantage tells us how much ‘extra’ reward we get from taking action a while in state s . The advantage bridges the gap between $Q(s, a)$ and $V(s)$ as $Q(s, a) = V(s) + A(s, a)$.

In the paper *Dueling Network Architectures for Deep Reinforcement Learning* <<https://arxiv.org/abs/1511.06581>> the network uses two heads, one outputs the value state and the other outputs the advantage. This leads to better training stability, faster convergence and overall better results. The V head outputs a single scalar (the state value), while the advantage head outputs a tensor equal to the size of the action space, containing an advantage value for each action in state s .

Changing the network architecture is not enough, we also need to ensure that the advantage mean is 0. This is done by subtracting the mean advantage from the Q value. This essentially pulls the mean advantage to 0.

$$Q(s, a) = V(s) + A(s, a) - 1/N * \sum_k (A(s, k))$$

Dueling DQN Benefits

- Ability to efficiently learn the state value function. In the dueling network, every Q update also updates the value stream, where as in DQN only the value of the chosen action is updated. This provides a better approximation of the values
- The differences between total Q values for a given state are quite small in relation to the magnitude of Q. The difference in the Q values between the best action and the second best action can be very small, while the average state value can be much larger. The differences in scale can introduce noise, which may lead to the greedy policy switching the priority of these actions. The separate estimators for state value and advantage makes the Dueling DQN robust to this type of scenario

Dueling DQN Results

The results below a noticeable improvement from the original DQN network.

Dueling DQN baseline: Pong

Similar to the results of the DQN baseline, the agent has a period where the number of steps per episodes increase as it begins to hold its own against the heuristic oppoent, but then the steps per episode quickly begins to drop as it gets better and starts to beat its opponent faster and faster. There is a noticable point at step ~250k where the agent goes from losing to winning.

As you can see by the total rewards, the dueling network's training progression is very stable and continues to trend upward until it finally plateaus.

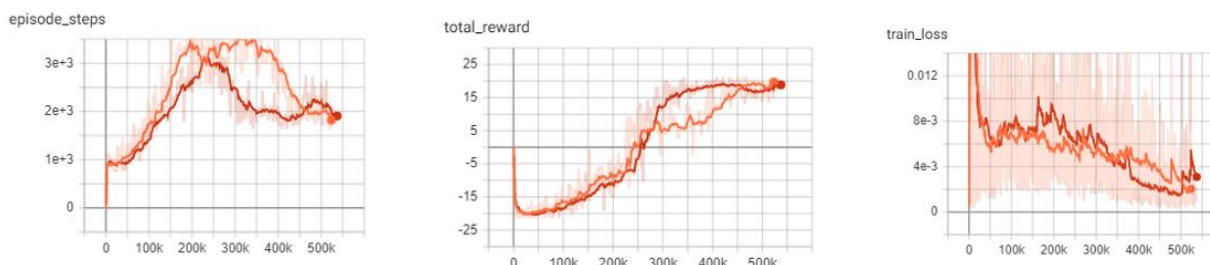


DQN vs Dueling DQN: Pong

In comparison to the base DQN, we see that the Dueling network's training is much more stable and is able to reach a score in the high teens faster than the DQN agent. Even though the Dueling network is more stable and out performs DQN early in training, by the end of training the two networks end up at the same point.

This could very well be due to the simplicity of the Pong environment.

- Orange: DQN
- Red: Dueling DQN



Example:

```
from pl_bolts.models.rl import DuelingDQN
dueling_dqn = DuelingDQN("PongNoFrameskip-v4")
trainer = Trainer()
trainer.fit(dueling_dqn)
```

```
class pl_bolts.models.rl.dueling_dqn_model.DuelingDQN(env, eps_start=1.0,
                                                    eps_end=0.02,
                                                    eps_last_frame=150000,
                                                    sync_rate=1000,
                                                    gamma=0.99,          learn-
                                                    ing_rate=0.0001,
                                                    batch_size=32,          re-
                                                    play_size=100000,
                                                    warm_start_size=10000,
                                                    avg_reward_len=100,
                                                    min_episode_reward=-
                                                    21,          seed=123,
                                                    batches_per_epoch=1000,
                                                    n_steps=1, **kwargs)
```

Bases: `pl_bolts.models.rl.dqn_model.DQN`

PyTorch Lightning implementation of `Dueling DQN`

Paper authors: Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas

Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.dueling_dqn_model import DuelingDQN
...
>>> model = DuelingDQN("PongNoFrameskip-v4")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: Currently only supports CPU and single GPU training with `distributed_backend=dp`

Parameters

- **env** (str) – gym environment tag
- **eps_start** (float) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (float) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (int) – the final frame in for the decrease of epsilon. At this frame `epsilon = eps_end`
- **sync_rate** (int) – the number of iterations between syncing up the target network with the train network
- **gamma** (float) – discount factor
- **learning_rate** (float) – learning rate
- **batch_size** (int) – size of minibatch pulled from the DataLoader
- **replay_size** (int) – total capacity of the replay buffer

- **warm_start_size** (int) – how many random steps through the environment to be carried out at the start of training to fill the buffer with a starting point
- **avg_reward_len** (int) – how many episodes to take into account when calculating the avg reward
- **min_episode_reward** (int) – the minimum score that can be achieved in an episode. Used for filling the avg buffer before training begins
- **seed** (int) – seed value for all RNG used
- **batches_per_epoch** (int) – number of batches per epoch
- **n_steps** (int) – size of n step look ahead

build_networks ()

Initializes the Dueling DQN train and target networks

Return type None

22.2.4 Noisy DQN

Noisy DQN model introduced in [Noisy Networks for Exploration](#) Paper authors: Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, Shane Legg

Original implementation by: [Donal Byrne](#)

Up until now the DQN agent uses a separate exploration policy, generally epsilon-greedy where start and end values are set for its exploration. *Noisy Networks For Exploration* <<https://arxiv.org/abs/1706.10295>> introduces a new exploration strategy by adding noise parameters to the weights of the fully connect layers which get updated during backpropagation of the network. The noise parameters drive the exploration of the network instead of simply taking random actions more frequently at the start of training and less frequently towards the end. The of authors of propose two ways of doing this.

During the optimization step a new set of noisy parameters are sampled. During training the agent acts according to the fixed set of parameters. At the next optimization step, the parameters are updated with a new sample. This ensures the agent always acts based on the parameters that are drawn from the current noise distribution.

The authors propose two methods of injecting noise to the network.

- 1) Independent Gaussian Noise: This injects noise per weight. For each weight a random value is taken from the distribution. Noise parameters are stored inside the layer and are updated during backpropagation. The output of the layer is calculated as normal.
- 2) Factorized Gaussian Noise: This injects nosier per input/ouput. In order to minimize the number of random values this method stores two random vectors, one with the size of the input and the other with the size of the output. Using these two vectors, a random matrix is generated for the layer by calculating the outer products of the vector

Noisy DQN Benefits

- Improved exploration function. Instead of just performing completely random actions, we add decreasing amount of noise and uncertainty to our policy allowing to explore while still utilising its policy.
- The fact that this method is automatically tuned means that we do not have to tune hyper parameters for epsilon-greedy!

Note: For now I have just implemented the Independent Gaussian as it has been reported there isn't much difference in results for these benchmark environments.

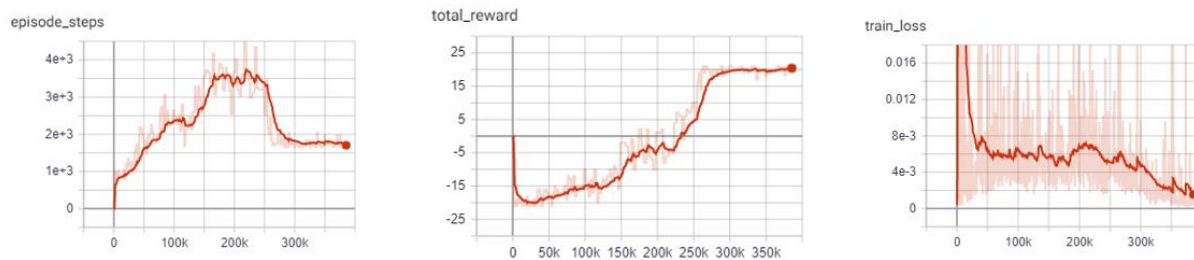
In order to update the basic DQN to a Noisy DQN we need to do the following

Noisy DQN Results

The results below improved stability and faster performance growth.

Noisy DQN baseline: Pong

Similar to the other improvements, the average score of the agent reaches positive numbers around the 250k mark and steadily increases till convergence.



DQN vs Noisy DQN: Pong

In comparison to the base DQN, the Noisy DQN is more stable and is able to converge on an optimal policy much faster than the original. It seems that the replacement of the epsilon-greedy strategy with network noise provides a better form of exploration.

- Orange: DQN
- Red: Noisy DQN



Example:

```
from pl_bolts.models.rl import NoisyDQN
noisy_dqn = NoisyDQN("PongNoFrameskip-v4")
trainer = Trainer()
trainer.fit(noisy_dqn)
```

```
class pl_bolts.models.rl.noisy_dqn_model.NoisyDQN(env, eps_start=1.0, eps_end=0.02,
                                                  eps_last_frame=150000,
                                                  sync_rate=1000, gamma=0.99,
                                                  learning_rate=0.0001,
                                                  batch_size=32, re-
                                                  play_size=100000,
                                                  warm_start_size=10000,
                                                  avg_reward_len=100,
                                                  min_episode_reward=-
                                                  21, seed=123,
                                                  batches_per_epoch=1000,
                                                  n_steps=1, **kwargs)
```

Bases: `pl_bolts.models.rl.dqn_model.DQN`

PyTorch Lightning implementation of Noisy DQN

Paper authors: Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, Shane Legg

Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.noisy_dqn_model import NoisyDQN
...
>>> model = NoisyDQN("PongNoFrameskip-v4")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: Currently only supports CPU and single GPU training with `distributed_backend=dp`

Parameters

- **env** `(str)` – gym environment tag
- **eps_start** `(float)` – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** `(float)` – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** `(int)` – the final frame in for the decrease of epsilon. At this frame `epsilon = eps_end`
- **sync_rate** `(int)` – the number of iterations between syncing up the target network with the train network
- **gamma** `(float)` – discount factor

- **learning_rate** (float) – learning rate
- **batch_size** (int) – size of minibatch pulled from the DataLoader
- **replay_size** (int) – total capacity of the replay buffer
- **warm_start_size** (int) – how many random steps through the environment to be carried out at the start of training to fill the buffer with a starting point
- **avg_reward_len** (int) – how many episodes to take into account when calculating the avg reward
- **min_episode_reward** (int) – the minimum score that can be achieved in an episode. Used for filling the avg buffer before training begins
- **seed** (int) – seed value for all RNG used
- **batches_per_epoch** (int) – number of batches per epoch
- **n_steps** (int) – size of n step look ahead

build_networks ()

Initializes the Noisy DQN train and target networks

Return type None

on_train_start ()

Set the agents epsilon to 0 as the exploration comes from the network

Return type None

train_batch ()

Contains the logic for generating a new batch of data to be passed to the DataLoader. This is the same function as the standard DQN except that we dont update epsilon as it is always 0. The exploration comes from the noisy network.

Return type Tuple[`Tensor`, `Tensor`, `Tensor`, `Tensor`, `Tensor`]

Returns yields a Experience tuple containing the state, action, reward, done and next_state.

22.2.5 N-Step DQN

N-Step DQN model introduced in [Learning to Predict by the Methods of Temporal Differences](#) Paper authors: Richard S. Sutton

Original implementation by: [Donal Byrne](#)

N Step DQN was introduced in [Learning to Predict by the Methods of Temporal Differences](#). This method improves upon the original DQN by updating our Q values with the expected reward from multiple steps in the future as opposed to the expected reward from the immediate next state. When getting the Q values for a state action pair using a single step which looks like this

$$Q(s_t, a_t) = r_t + \gamma \max_a Q(s_{t+1}, a_{t+1})$$

but because the Q function is recursive we can continue to roll this out into multiple steps, looking at the expected return for each step into the future.

$$Q(s_t, a_t) = r_t + \gamma r_{t+1} + \gamma^2 \max_{a'} Q(s_{t+2}, a')$$

The above example shows a 2-Step look ahead, but this could be rolled out to the end of the episode, which is just Monte Carlo learning. Although we could just do a monte carlo update and look forward to the end of the episode,

it wouldn't be a good idea. Every time we take another step into the future, we are basing our approximation off our current policy. For a large portion of training, our policy is going to be less than optimal. For example, at the start of training, our policy will be in a state of high exploration, and will be little better than random.

Note: For each rollout step you must scale the discount factor accordingly by the number of steps. As you can see from the equation above, the second gamma value is to the power of 2. If we rolled this out one step further, we would use gamma to the power of 3 and so.

So if we are approximating future rewards off a bad policy, chances are those approximations are going to be pretty bad and every time we unroll our update equation, the worse it will get. The fact that we are using an off policy method like DQN with a large replay buffer will make this even worse, as there is a high chance that we will be training on experiences using an old policy that was worse than our current policy.

So we need to strike a balance between looking far enough ahead to improve the convergence of our agent, but not so far that our updates become unstable. In general, small values of 2-4 work best.

N-Step Benefits

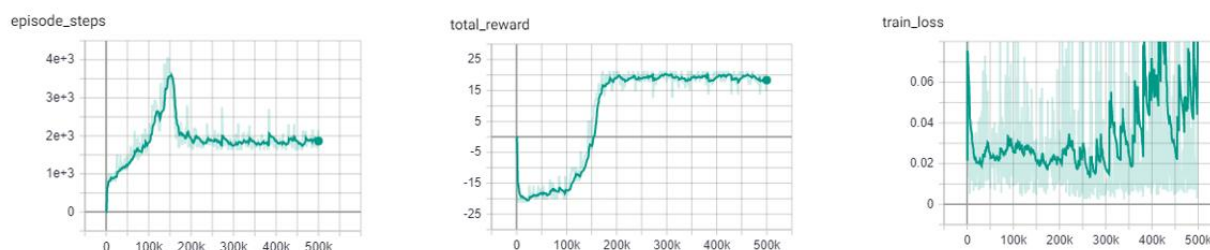
- Multi-Step learning is capable of learning faster than typical 1 step learning methods.
- Note that this method introduces a new hyperparameter n . Although $n=4$ is generally a good starting point and provides good results across the board.

N-Step Results

As expected, the N-Step DQN converges much faster than the standard DQN, however it also adds more instability to the loss of the agent. This can be seen in the following experiments.

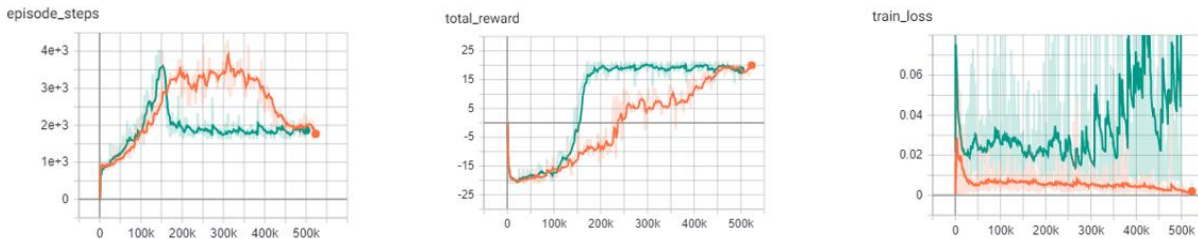
N-Step DQN: Pong

The N-Step DQN shows the greatest increase in performance with respect to the other DQN variations. After less than 150k steps the agent begins to consistently win games and achieves the top score after ~170K steps. This is reflected in the sharp peak of the total episode steps and of course, the total episode rewards.



DQN vs N-Step DQN: Pong

This improvement is shown in stark contrast to the base DQN, which only begins to win games after 250k steps and requires over twice as many steps (450k) as the N-Step agent to achieve the high score of 21. One important thing to notice is the large increase in the loss of the N-Step agent. This is expected as the agent is building its expected reward off approximations of the future states. The larger the size of N , the greater the instability. Previous literature, listed below, shows the best results for the Pong environment with an N step between 3-5. For these experiments I opted with an N step of 4.



Example:

```
from pl_bolts.models.rl import DQN
n_step_dqn = DQN("PongNoFrameskip-v4", n_steps=4)
trainer = Trainer()
trainer.fit(n_step_dqn)
```

22.2.6 Prioritized Experience Replay DQN

Double DQN model introduced in [Prioritized Experience Replay](#) Paper authors: Tom Schaul, John Quan, Ioannis Antonoglou, David Silver

Original implementation by: [Donal Byrne](#)

The standard DQN uses a buffer to break up the correlation between experiences and uniform random samples for each batch. Instead of just randomly sampling from the buffer prioritized experience replay (PER) prioritizes these samples based on training loss. This concept was introduced in the paper [Prioritized Experience Replay](#)

Essentially we want to train more on the samples that surprise the agent.

The priority of each sample is defined below where

$$P(i) = P_i^\alpha / \sum_k P_k^\alpha$$

where p_i is the priority of the i th sample in the buffer and α is the number that shows how much emphasis we give to the priority. If $\alpha = 0$, our sampling will become uniform as in the classic DQN method. Larger values for α put more stress on samples with higher priority

Its important that new samples are set to the highest priority so that they are sampled soon. This however introduces bias to new samples in our dataset. In order to compensate for this bias, the value of the weight is defined as

$$w_i = (N \cdot P(i))^{-\beta}$$

Where β is a hyper parameter between 0-1. When β is 1 the bias is fully compensated. However authors noted that in practice it is better to start β with a small value near 0 and slowly increase it to 1.

PER Benefits

- The benefits of this technique are that the agent sees more samples that it struggled with and gets more chances to improve upon it.

Memory Buffer

First step is to replace the standard experience replay buffer with the prioritized experience replay buffer. This is pretty large (100+ lines) so I wont go through it here. There are two buffers implemented. The first is a naive list based buffer found in `memory.PERBuffer` and the second is more efficient buffer using a Sum Tree datastructure.

The list based version is simpler, but has a sample complexity of $O(N)$. The Sum Tree in comparison has a complexity of $O(1)$ for sampling and $O(\log N)$ for updating priorities.

Update loss function

The next thing we do is to use the sample weights that we get from PER. Add the following code to the end of the loss function. This applies the weights of our sample to the batch loss. Then we return the mean loss and weighted loss for each datum, with the addition of a small epsilon value.

PER Results

The results below show improved stability and faster performance growth.

PER DQN: Pong

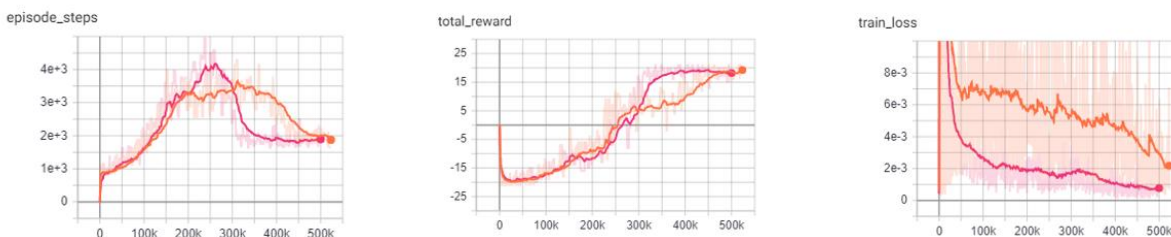
Similar to the other improvements, we see that PER improves the stability of the agents training and shows to converge on an optimal policy faster.



DQN vs PER DQN: Pong

In comparison to the base DQN, the PER DQN does show improved stability and performance. As expected, the loss of the PER DQN is significantly lower. This is the main objective of PER by focusing on experiences with high loss.

It is important to note that loss is not the only metric we should be looking at. Although the agent may have very low loss during training, it may still perform poorly due to lack of exploration.



- Orange: DQN
- Pink: PER DQN

Example:

```
from pl_bolts.models.rl import PERDQN
per_dqn = PERDQN("PongNoFrameskip-v4")
trainer = Trainer()
trainer.fit(per_dqn)
```

```
class pl_bolts.models.rl.per_dqn_model.PERDQN(env, eps_start=1.0, eps_end=0.02,
                                              eps_last_frame=150000,
                                              sync_rate=1000, gamma=0.99,
                                              learning_rate=0.0001,
                                              batch_size=32, replay_size=100000,
                                              warm_start_size=10000,
                                              avg_reward_len=100,
                                              min_episode_reward=-21, seed=123,
                                              batches_per_epoch=1000, n_steps=1,
                                              **kwargs)
```

Bases: `pl_bolts.models.rl.dqn_model.DQN`

PyTorch Lightning implementation of DQN With Prioritized Experience Replay

Paper authors: Tom Schaul, John Quan, Ioannis Antonoglou, David Silver

Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.per_dqn_model import PERDQN
...
>>> model = PERDQN("PongNoFrameskip-v4")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note:

This example is based on: https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter08/05_dqn_prio_replay.py

Note: Currently only supports CPU and single GPU training with `distributed_backend=dp`

Parameters

- `env` (str) – gym environment tag
- `eps_start` (float) – starting value of epsilon for the epsilon-greedy exploration
- `eps_end` (float) – final value of epsilon for the epsilon-greedy exploration
- `eps_last_frame` (int) – the final frame in for the decrease of epsilon. At this frame `epsilon = eps_end`
- `sync_rate` (int) – the number of iterations between syncing up the target network with the train network
- `gamma` (float) – discount factor
- `learning_rate` (float) – learning rate
- `batch_size` (int) – size of minibatch pulled from the DataLoader

- **replay_size** (int) – total capacity of the replay buffer
- **warm_start_size** (int) – how many random steps through the environment to be carried out at the start of training to fill the buffer with a starting point
- **avg_reward_len** (int) – how many episodes to take into account when calculating the avg reward
- **min_episode_reward** (int) – the minimum score that can be achieved in an episode. Used for filling the avg buffer before training begins
- **seed** (int) – seed value for all RNG used
- **batches_per_epoch** (int) – number of batches per epoch
- **n_steps** (int) – size of n step look ahead

train_batch ()

Contains the logic for generating a new batch of data to be passed to the DataLoader

Return type Tuple[`Tensor`, `Tensor`, `Tensor`, `Tensor`, `Tensor`]

Returns yields a Experience tuple containing the state, action, reward, done and next_state.

training_step (batch, _)

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch recieved

Parameters

- **batch** – current mini batch of replay data
- **_** – batch number, not used

Return type `OrderedDict`

Returns Training loss and log metrics

22.3 Policy Gradient Models

The following models are based on Policy Gradients. Unlike the Q learning models shown before, Policy based models do not try and learn the specific values of state or state action pairs. Instead it cuts out the middle man and directly learns the policy distribution. In Policy Gradient models we update our network parameters in the direction suggested by our policy gradient in order to find a policy that produces the highest results.

Policy Gradient Key Points:

- Outputs a distribution of actions instead of discrete Q values
- Optimizes the policy directly, instead of indirectly through the optimization of Q values
- The policy distribution of actions allows the model to handle more complex action spaces, such as continuous actions
- The policy distribution introduces stochasticity, providing natural exploration to the model
- The policy distribution provides a more stable update as a change in weights will only change the total distribution slightly, as opposed to changing weights based on the Q value of state S will change all Q values with similar states.
- Policy gradients tend to converge faster, however they are not as sample efficient and generally require more interactions with the environment.

22.3.1 REINFORCE

REINFORCE model introduced in [Policy Gradient Methods For Reinforcement Learning With Function Approximation](#) Paper authors: Richard S. Sutton, David McAllester, Satinder Singh, Yishay Mansour

Original implementation by: [Donal Byrne](#)

REINFORCE is one of the simplest forms of the Policy Gradient method of RL. This method uses a Monte Carlo rollout, where its steps through entire episodes of the environment to build up trajectories computing the total rewards. The algorithm is as follows:

1. Initialize our network.
2. Play N full episodes saving the transitions through the environment.
3. For every step t in each episode k we calculate the discounted reward of the subsequent steps.

$$Q_{k,t} = \sum_{i=0} \gamma^i r_i$$

4. Calculate the loss for all transitions.

$$L = - \sum_{k,t} Q_{k,t} \log(\pi(S_{k,t}, A_{k,t}))$$

5. Perform SGD on the loss and repeat.

What this loss function is saying is simply that we want to take the log probability of action A at state S given our policy (network output). This is then scaled by the discounted reward that we calculated in the previous step. We then take the negative of our sum. This is because the loss is minimized during SGD, but we want to maximize our policy.

Note: The current implementation does not actually wait for the batch episodes the complete every time as we pass in a fixed batch size. For the time being we simply use a large batch size to accomodate this. This approach still works well for simple tasks as it still manages to get an accurate Q value by using a large batch size, but it is not as accurate or completely correct. This will be updated in a later version.

REINFORCE Benefits

- Simple and straightforward
- Computationally more efficient for simple tasks such as Cartpole than the Value Based methods.

REINFORCE Results

Hyperparameters:

- Batch Size: 800
- Learning Rate: 0.01
- Episodes Per Batch: 4
- Gamma: 0.99

TODO: Add results graph

Example:

```
from pl_bolts.models.rl import Reinforce
reinforce = Reinforce("CartPole-v0")
trainer = Trainer()
trainer.fit(reinforce)
```

```
class pl_bolts.models.rl.reinforce_model.Reinforce(env, gamma=0.99, lr=0.01,
                                                    batch_size=8, n_steps=10,
                                                    avg_reward_len=100,
                                                    entropy_beta=0.01,
                                                    epoch_len=1000,
                                                    num_batch_episodes=4,
                                                    **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of **REINFORCE** Paper authors: Richard S. Sutton, David McAllester, Satinder Singh, Yishay Mansour Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.reinforce_model import Reinforce
...
>>> model = Reinforce("CartPole-v0")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: This example is based on: https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter11/02_cartpole_reinforce.py

Note: Currently only supports CPU and single GPU training with *distributed_backend=dp*

Parameters

- **env** (str) – gym environment tag
- **gamma** (float) – discount factor
- **lr** (float) – learning rate
- **batch_size** (int) – size of minibatch pulled from the DataLoader
- **n_steps** (int) – number of stakes per discounted experience
- **entropy_beta** (float) – entropy coefficient
- **epoch_len** (int) – how many batches before pseudo epoch
- **num_batch_episodes** (int) – how many episodes to rollout for each batch of training
- **avg_reward_len** (int) – how many episodes to take into account when calculating the avg reward

static add_model_specific_args (*arg_parser*)
Adds arguments for DQN model

Note: These params are fine tuned for Pong env.

Parameters *arg_parser* – the current argument parser to add to

Return type `ArgumentParser`

Returns *arg_parser* with model specific args added

calc_qvals (*rewards*)
Calculate the discounted rewards of all rewards in list

Parameters *rewards* (`List[float]`) – list of rewards from latest batch

Return type `List[float]`

Returns list of discounted rewards

configure_optimizers ()
Initialize Adam optimizer

Return type `List[Optimizer]`

discount_rewards (*experiences*)
Calculates the discounted reward over N experiences

Parameters *experiences* (`Tuple[Experience]`) – Tuple of Experience

Return type `float`

Returns total discounted reward

forward (*x*)
Passes in a state x through the network and gets the q_values of each action as an output

Parameters *x* (`Tensor`) – environment state

Return type `Tensor`

Returns q values

get_device (*batch*)
Retrieve device currently being used by minibatch

Return type `str`

train_batch ()
Contains the logic for generating a new batch of data to be passed to the DataLoader

Yields yields a tuple of Lists containing tensors for states, actions and rewards of the batch.

Return type `Tuple[List[Tensor], List[Tensor], List[Tensor]]`

train_dataloader ()
Get train loader

Return type `DataLoader`

training_step (*batch*, *_*)
Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch recieved

Parameters

- `batch` (Tuple[`Tensor`, `Tensor`]) – current mini batch of replay data
- `_` – batch number, not used

Return type `OrderedDict`

Returns Training loss and log metrics

22.3.2 Vanilla Policy Gradient

Vanilla Policy Gradient model introduced in [Policy Gradient Methods For Reinforcement Learning With Function Approximation](#) Paper authors: Richard S. Sutton, David McAllester, Satinder Singh, Yishay Mansour

Original implementation by: [Donal Byrne](#)

Vanilla Policy Gradient (VPG) expands upon the REINFORCE algorithm and improves some of its major issues. The major issue with REINFORCE is that it has high variance. This can be improved by subtracting a baseline value from the Q values. For this implementation we use the average reward as our baseline.

Although Policy Gradients are able to explore naturally due to the stochastic nature of the model, the agent can still frequently be stuck in a local optima. In order to improve this, VPG adds an entropy term to improve exploration.

$$H(\pi) = - \sum \pi(a|s) \log \pi(a|s)$$

To further control the amount of additional entropy in our model we scale the entropy term by a small beta value. The scaled entropy is then subtracted from the policy loss.

VPG Benefits

- Addition of the baseline reduces variance in the model
- Improved exploration due to entropy bonus

VPG Results

Hyperparameters:

- Batch Size: 8
- Learning Rate: 0.001
- N Steps: 10
- N environments: 4
- Entropy Beta: 0.01
- Gamma: 0.99

Example:

```
from pl_bolts.models.rl import VanillaPolicyGradient
vpg = VanillaPolicyGradient("CartPole-v0")
trainer = Trainer()
trainer.fit(vpg)
```

```
class pl_bolts.models.rl.vanilla_policy_gradient_model.VanillaPolicyGradient (env,
                                                                              gamma=0.99,
                                                                              lr=0.01,
                                                                              batch_size=8,
                                                                              n_steps=10,
                                                                              avg_reward_len=10,
                                                                              entropy_beta=0.01,
                                                                              epoch_len=1000,
                                                                              **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [Vanilla Policy Gradient](#)

Paper authors: Richard S. Sutton, David McAllester, Satinder Singh, Yishay Mansour

Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.vanilla_policy_gradient_model import _
    ↪ VanillaPolicyGradient
    ...
>>> model = VanillaPolicyGradient("CartPole-v0")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: This example is based on: https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter11/04_cartpole_pg.py

Note: Currently only supports CPU and single GPU training with `distributed_backend=dp`

Parameters

- `env` (str) – gym environment tag
- `gamma` (float) – discount factor
- `lr` (float) – learning rate
- `batch_size` (int) – size of minibatch pulled from the DataLoader
- `batch_episodes` – how many episodes to rollout for each batch of training
- `entropy_beta` (float) – dictates the level of entropy per batch
- `avg_reward_len` (int) – how many episodes to take into account when calculating the avg reward
- `epoch_len` (int) – how many batches before pseudo epoch

static add_model_specific_args (*arg_parser*)
 Adds arguments for DQN model

Note: These params are fine tuned for Pong env.

Parameters *arg_parser* – the current argument parser to add to

Return type `ArgumentParser`

Returns *arg_parser* with model specific args added

compute_returns (*rewards*)
 Calculate the discounted rewards of the batched rewards

Parameters *rewards* – list of batched rewards

Returns list of discounted rewards

configure_optimizers ()
 Initialize Adam optimizer

Return type `List[Optimizer]`

forward (*x*)
 Passes in a state *x* through the network and gets the q_values of each action as an output

Parameters *x* (`Tensor`) – environment state

Return type `Tensor`

Returns q values

get_device (*batch*)
 Retrieve device currently being used by minibatch

Return type `str`

loss (*states*, *actions*, *scaled_rewards*)
 Calculates the loss for VPG

Parameters

- *states* – batched states
- *actions* – batch actions
- *scaled_rewards* – batched Q values

Return type `Tensor`

Returns loss for the current batch

train_batch ()
 Contains the logic for generating a new batch of data to be passed to the DataLoader

Return type `Tuple[List[Tensor], List[Tensor], List[Tensor]]`

Returns yields a tuple of Lists containing tensors for states, actions and rewards of the batch.

train_dataloader ()
 Get train loader

Return type `DataLoader`

training_step (*batch*, *_*)

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch recieved

Parameters

- **batch** `Tensor` (Tuple[`Tensor`, `Tensor`]) – current mini batch of replay data
- **_** – batch number, not used

Return type `OrderedDict`

Returns Training loss and log metrics

SELF-SUPERVISED LEARNING

This bolts module houses a collection of all self-supervised learning models.

Self-supervised learning extracts representations of an input by solving a pretext task. In this package, we implement many of the current state-of-the-art self-supervised algorithms.

Self-supervised models are trained with unlabeled datasets

23.1 Use cases

Here are some use cases for the self-supervised package.

23.1.1 Extracting image features

The models in this module are trained unsupervised and thus can capture better image representations (features).

In this example, we'll load a resnet 18 which was pretrained on imagenet using CPC as the pretext task.

```
from pl_bolts.models.self_supervised import SimCLR

# load resnet50 pretrained using SimCLR on imagenet
weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/simclr/bolts_
↳simclr_imagenet/simclr_imagenet.ckpt'
simclr = SimCLR.load_from_checkpoint(weight_path, strict=False)

simclr_resnet50 = simclr.encoder
simclr_resnet50.eval()
```

This means you can now extract image representations that were pretrained via unsupervised learning.

Example:

```
my_dataset = SomeDataset()
for batch in my_dataset:
    x, y = batch
    out = simclr_resnet50(x)
```

23.1.2 Train with unlabeled data

These models are perfect for training from scratch when you have a huge set of unlabeled images

```
from pl_bolts.models.self_supervised import SimCLR
from pl_bolts.models.self_supervised.simclr import SimCLREvalDataTransform, \
    SimCLRTrainDataTransform

train_dataset = MyDataset(transforms=SimCLRTrainDataTransform())
val_dataset = MyDataset(transforms=SimCLREvalDataTransform())

# simclr needs a lot of compute!
model = SimCLR()
trainer = Trainer(tpu_cores=128)
trainer.fit(
    model,
    DataLoader(train_dataset),
    DataLoader(val_dataset),
)
```

23.1.3 Research

Mix and match any part, or subclass to create your own new method

```
from pl_bolts.models.self_supervised import CPCV2
from pl_bolts.losses.self_supervised_learning import FeatureMapContrastiveTask

amdim_task = FeatureMapContrastiveTask(comparisons='01, 11, 02', bidirectional=True)
model = CPCV2(contrastive_task=amdim_task)
```

23.2 Contrastive Learning Models

Contrastive self-supervised learning (CSL) is a self-supervised learning approach where we generate representations of instances such that similar instances are near each other and far from dissimilar ones. This is often done by comparing triplets of positive, anchor and negative representations.

In this section, we list Lightning implementations of popular contrastive learning approaches.

23.2.1 AMDIM

```
class pl_bolts.models.self_supervised.AMDIM(datamodule='cifar10',
                                             encoder='amdim_encoder',
                                             contrastive_task=torch.nn.Module,
                                             image_channels=3,
                                             image_height=32,
                                             encoder_feature_dim=320,
                                             embedding_fx_dim=1280,
                                             conv_block_depth=10,
                                             use_bn=False,
                                             tclip=20.0,
                                             learning_rate=0.0002,
                                             data_dir="",
                                             num_classes=10,
                                             batch_size=200,
                                             num_workers=16, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [Augmented Multiscale Deep InfoMax \(AMDIM\)](#).

Paper authors: Philip Bachman, R Devon Hjelm, William Buchwalter.

Model implemented by: [William Falcon](#)

This code is adapted to Lightning using the original author repo ([the original repo](#)).

Example

```
>>> from pl_bolts.models.self_supervised import AMDIM
...
>>> model = AMDIM(encoder='resnet18')
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Parameters

- **datamodule** `(Union[str, LightningDataModule])` – A LightningDatamodule
- **encoder** `(Union[str, Module, LightningModule])` – an encoder string or model
- **image_channels** `(int)` – 3
- **image_height** `(int)` – pixels
- **encoder_feature_dim** `(int)` – Called *ndf* in the paper, this is the representation size for the encoder.
- **embedding_fx_dim** `(int)` – Output dim of the embedding function (*nrkhs* in the paper) (Reproducing Kernel Hilbert Spaces).
- **conv_block_depth** `(int)` – Depth of each encoder block,
- **use_bn** `(bool)` – If true will use batchnorm.
- **tclip** `(int)` – soft clipping non-linearity to the scores after computing the regularization term and before computing the log-softmax. This is the ‘second trick’ used in the paper
- **learning_rate** `(int)` – The learning rate
- **data_dir** `(str)` – Where to store data
- **num_classes** `(int)` – How many classes in the dataset
- **batch_size** `(int)` – The batch size

23.2.2 BYOL

```
class pl_bolts.models.self_supervised.BYOL(num_classes, learning_rate=0.2,
                                             weight_decay=1.5e-06, input_height=32,
                                             batch_size=32, num_workers=0,
                                             warmup_epochs=10, max_epochs=1000,
                                             **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [Bootstrap Your Own Latent \(BYOL\)](#)

Paper authors: Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre H. Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, Bilal Piot, Koray Kavukcuoglu, Rémi Munos, Michal Valko.

Model implemented by:

- [Annika Brundyn](#)

Warning: Work in progress. This implementation is still being verified.

TODOs:

- verify on CIFAR-10
- verify on STL-10
- pre-train on imagenet

Example:

```
model = BYOL(num_classes=10)

dm = CIFAR10DataModule(num_workers=0)
dm.train_transforms = SimCLRTrainDataTransform(32)
dm.val_transforms = SimCLREvalDataTransform(32)

trainer = pl.Trainer()
trainer.fit(model, datamodule=dm)
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

CLI command:

```
# cifar10
python byol_module.py --gpus 1

# imagenet
python byol_module.py
  --gpus 8
  --dataset imagenet2012
  --data_dir /path/to/imagenet/
  --meta_dir /path/to/folder/with/meta.bin/
  --batch_size 32
```

Parameters

- **datamodule** – The datamodule
- **learning_rate** (float) – the learning rate
- **weight_decay** (float) – optimizer weight decay
- **input_height** (int) – image input height
- **batch_size** (int) – the batch size
- **num_workers** (int) – number of workers

- `warmup_epochs` (int) – num of epochs for scheduler warm up
- `max_epochs` (int) – max epochs for scheduler

23.2.3 CPC (V2)

PyTorch Lightning implementation of [Data-Efficient Image Recognition with Contrastive Predictive Coding](#)

Paper authors: (Olivier J. Hénaff, Aravind Srinivas, Jeffrey De Fauw, Ali Razavi, Carl Doersch, S. M. Ali Eslami, Aaron van den Oord).

Model implemented by:

- William Falcon
- Tullie Murrell

To Train:

```
import pytorch_lightning as pl
from pl_bolts.models.self_supervised import CPCV2
from pl_bolts.datamodules import CIFAR10DataModule
from pl_bolts.models.self_supervised.cpc import (
    CPCTrainTransformsCIFAR10, CPCEvalTransformsCIFAR10)

# data
dm = CIFAR10DataModule(num_workers=0)
dm.train_transforms = CPCTrainTransformsCIFAR10()
dm.val_transforms = CPCEvalTransformsCIFAR10()

# model
model = CPCV2()

# fit
trainer = pl.Trainer()
trainer.fit(model, datamodule=dm)
```

To finetune:

```
python cpc_finetuner.py
    --ckpt_path path/to/checkpoint.ckpt
    --dataset cifar10
    --gpus 1
```

CIFAR-10 and STL-10 baselines

CPCv2 does not report baselines on CIFAR-10 and STL-10 datasets. Results in table are reported from the [YADIM](#) paper.

Table 1: CPCv2 implementation results

Dataset	test acc	Encoder	Optimizer	Batch	Epochs	Hardware	LR
CIFAR-10	84.52	CPCresnet101	Adam	64	1000 (upto 24 hours)	1 V100 (32GB)	4e-5
STL-10	78.36	CPCresnet101	Adam	144	1000 (upto 72 hours)	4 V100 (32GB)	1e-4
ImageNet	54.82	CPCresnet101	Adam	3072	1000 (upto 21 days)	64 V100 (32GB)	4e-5

CIFAR-10 pretrained model:

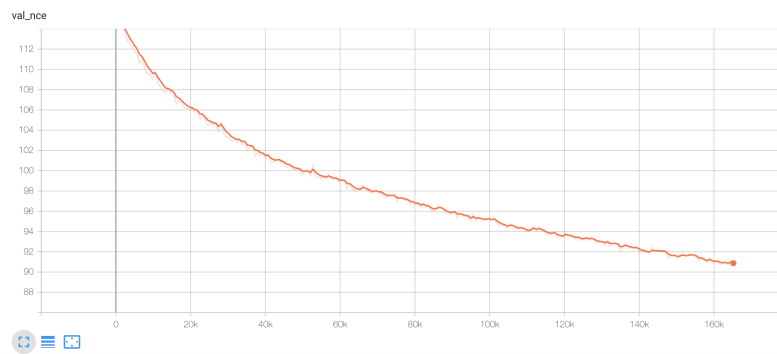
```
from pl_bolts.models.self_supervised import CPCV2

weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/cpc/cpc-cifar10-v4-
→exp3/epoch%3D474.ckpt'
cpc_v2 = CPCV2.load_from_checkpoint(weight_path, strict=False)

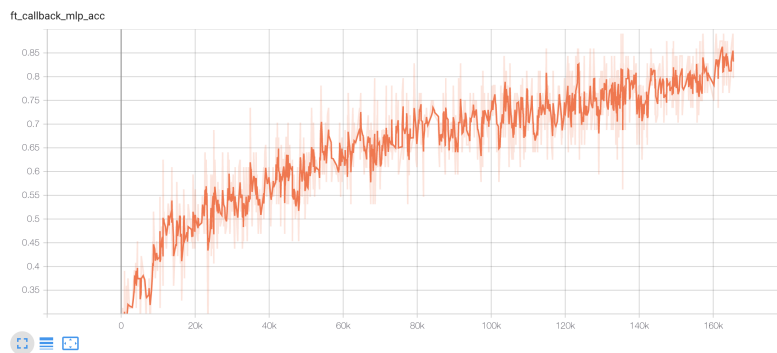
cpc_v2.freeze()
```

- Tensorboard for CIFAR10

Pre-training:



Fine-tuning:



STL-10 pretrained model:

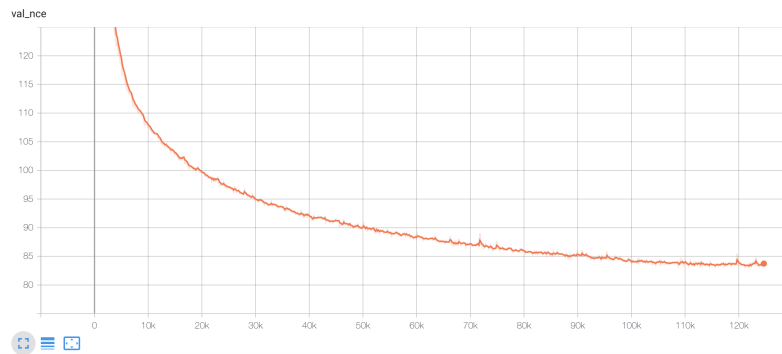
```
from pl_bolts.models.self_supervised import CPCV2

weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/cpc/cpc-stl10-v0-
↳exp3/epoch%3D624.ckpt'
cpc_v2 = CPCV2.load_from_checkpoint(weight_path, strict=False)

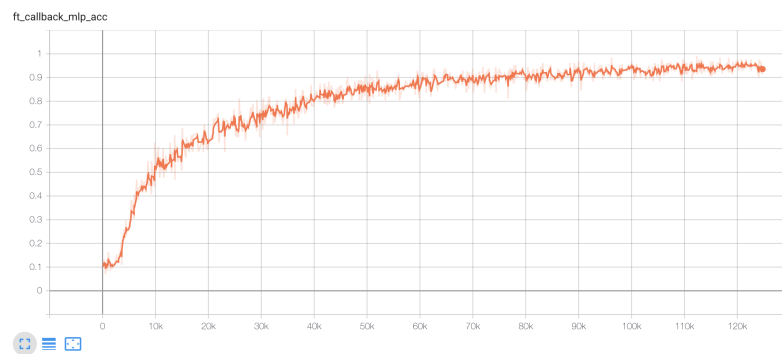
cpc_v2.freeze()
```

- Tensorboard for STL10

Pre-training:



Fine-tuning:



CPCV2 API

```
class pl_bolts.models.self_supervised.CPCV2(encoder_name='cpc_encoder',
                                             patch_size=8, patch_overlap=4, on-
                                             line_ft=True, task='cpc', num_workers=4,
                                             num_classes=10, learning_rate=0.0001,
                                             pretrained=None, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Parameters

- **encoder_name** `(str)` – A string for any of the resnets in torchvision, or the original CPC encoder, or a custom nn.Module encoder
- **patch_size** `(int)` – How big to make the image patches
- **patch_overlap** `(int)` – How much overlap each patch should have
- **online_ft** `(bool)` – If True, enables a 1024-unit MLP to fine-tune online
- **task** `(str)` – Which self-supervised task to use ('cpc', 'amdin', etc...)
- **num_workers** `(int)` – number of dataloader workers
- **num_classes** `(int)` – number of classes
- **learning_rate** `(float)` – learning rate
- **pretrained** `(Optional[str])` – If true, will use the weights pretrained (using CPC) on Imagenet

23.2.4 Moco (V2)

```
class pl_bolts.models.self_supervised.MocoV2(base_encoder='resnet18', emb_dim=128,
                                              num_negatives=65536, en-
                                              coder_momentum=0.999, soft-
                                              max_temperature=0.07, learn-
                                              ing_rate=0.03, momentum=0.9,
                                              weight_decay=0.0001, data_dir='.',
                                              batch_size=256, use_mlp=False,
                                              num_workers=8, *args, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [Moco](#)

Paper authors: Xinlei Chen, Haoqi Fan, Ross Girshick, Kaiming He.

Code adapted from [facebookresearch/moco](#) to Lightning by:

- [William Falcon](#)

Example:: `from pl_bolts.models.self_supervised import MocoV2 model = MocoV2() trainer = Trainer() trainer.fit(model)`

CLI command:

```
# cifar10
python moco2_module.py --gpus 1

# imagenet
```

(continues on next page)

(continued from previous page)

```
python moco2_module.py
--gpus 8
--dataset imagenet2012
--data_dir /path/to/imagenet/
--meta_dir /path/to/folder/with/meta.bin/
--batch_size 32
```

Parameters

- **base_encoder** (Union[str, Module]) – torchvision model name or torch.nn.Module
- **emb_dim** (int) – feature dimension (default: 128)
- **num_negatives** (int) – queue size; number of negative keys (default: 65536)
- **encoder_momentum** (float) – moco momentum of updating key encoder (default: 0.999)
- **softmax_temperature** (float) – softmax temperature (default: 0.07)
- **learning_rate** (float) – the learning rate
- **momentum** (float) – optimizer momentum
- **weight_decay** (float) – optimizer weight decay
- **datamodule** – the DataModule (train, val, test dataloaders)
- **data_dir** (str) – the directory to store data
- **batch_size** (int) – batch size
- **use_mlp** (bool) – add an mlp to the encoders
- **num_workers** (int) – workers for the loaders

forward (img_q, img_k)

Input: im_q: a batch of query images im_k: a batch of key images

Output: logits, targets

init_encoders (base_encoder)

Override to add your own encoders

23.2.5 SimCLR

PyTorch Lightning implementation of [SimCLR](#)

Paper authors: Ting Chen, Simon Kornblith, Mohammad Norouzi, Geoffrey Hinton.

Model implemented by:

- [William Falcon](#)
- [Tullie Murrell](#)
- [Ananya Harsh Jha](#)

To Train:

```

import pytorch_lightning as pl
from pl_bolts.models.self_supervised import SimCLR
from pl_bolts.datamodules import CIFAR10DataModule
from pl_bolts.models.self_supervised.simclr.transforms import (
    SimCLREvalDataTransform, SimCLRTrainDataTransform)

# data
dm = CIFAR10DataModule(num_workers=0)
dm.train_transforms = SimCLRTrainDataTransform(32)
dm.val_transforms = SimCLREvalDataTransform(32)

# model
model = SimCLR(num_samples=dm.num_samples, batch_size=dm.batch_size, dataset='cifar10
↪')

# fit
trainer = pl.Trainer()
trainer.fit(model, datamodule=dm)

```

CIFAR-10 baseline

Table 2: Cifar-10 implementation results

Implementation	test acc	Encoder	Optimizer	Batch	Epochs	Hardware	LR
Original	~94.00	resnet50	LARS	2048	800	TPUs	1.0/1.5
Ours	88.50	resnet50	LARS-SGD	2048	800 (4 hours)	8 V100 (16GB)	1.5

CIFAR-10 pretrained model:

```

from pl_bolts.models.self_supervised import SimCLR

weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/simclr/bolts_
↪simclr_imagenet/simclr_imagenet.ckpt'
simclr = SimCLR.load_from_checkpoint(weight_path, strict=False)

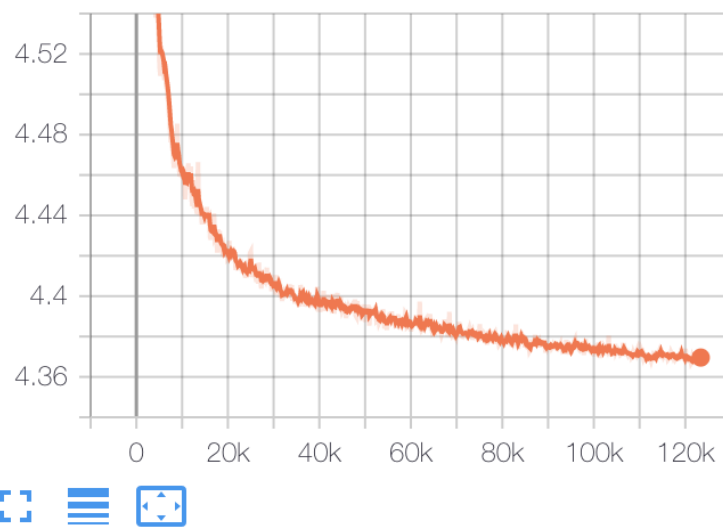
simclr.freeze()

```

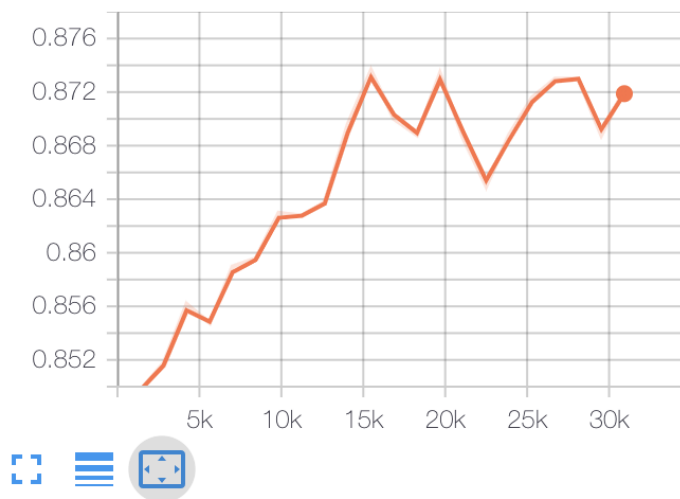
Pre-training:

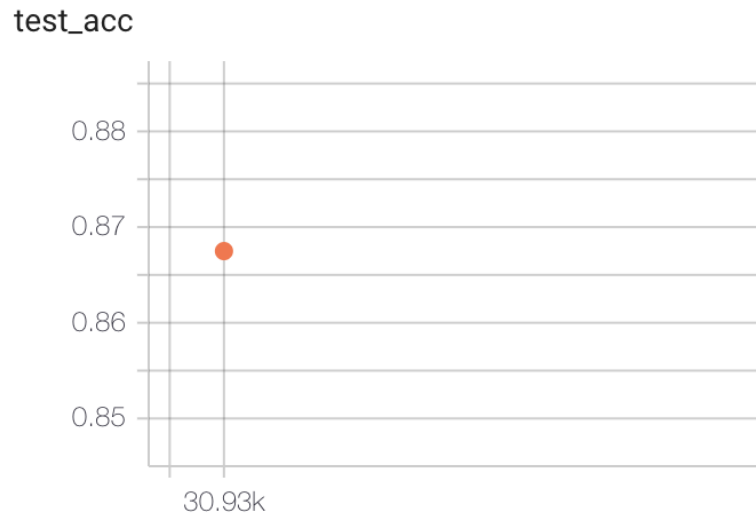
Fine-tuning (Single layer MLP, 1024 hidden units):

avg_val_loss



val_acc





To reproduce:

```
# pretrain
python simclr_module.py
    --gpus 8
    --dataset cifar10
    --batch_size 256
    -- num_workers 16
    --optimizer sgd
    --learning_rate 1.5
    --lars_wrapper
    --exclude_bn_bias
    --max_epochs 800
    --online_ft

# finetune
python simclr_finetuner.py
    --gpus 4
    --ckpt_path path/to/simclr/ckpt
    --dataset cifar10
    --batch_size 64
    --num_workers 8
    --learning_rate 0.3
    --num_epochs 100
```


Imagenet baseline for SimCLR

Table 3: Cifar-10 implementation results

Implementation	test acc	Encoder	Optimizer	Batch	Epochs	Hardware	LR
Original	~69.3	resnet50	LARS	4096	800	TPUs	4.8
Ours	68.4	resnet50	LARS-SGD	4096	800	64 V100 (16GB)	4.8

Imagenet pretrained model:

```
from pl_bolts.models.self_supervised import SimCLR

weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/simclr/bolts_
↳simclr_imagenet/simclr_imagenet.ckpt'
simclr = SimCLR.load_from_checkpoint(weight_path, strict=False)

simclr.freeze()
```

To reproduce:

```
# pretrain
python simclr_module.py
    --dataset imagenet
    --data_path path/to/imagenet

# finetune
python simclr_fineturner.py
    --gpus 8
    --ckpt_path path/to/simclr/ckpt
    --dataset imagenet
    --data_dir path/to/imagenet/dataset
    --batch_size 256
    --num_workers 16
    --learning_rate 0.8
    --nesterov True
    --num_epochs 90
```

SimCLR API

```
class pl_bolts.models.self_supervised.SimCLR(gpus, num_samples, batch_size,
                                             dataset, num_nodes=1, arch='resnet50',
                                             hidden_mlp=2048, feat_dim=128,
                                             warmup_epochs=10, max_epochs=100,
                                             temperature=0.1, first_conv=True,
                                             maxpool1=True, optimizer='adam',
                                             lars_wrapper=True, ex-
                                             clude_bn_bias=False, start_lr=0.0,
                                             learning_rate=0.001, final_lr=0.0,
                                             weight_decay=1e-06, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Parameters

- **batch_size** `(int)` – the batch size
- **num_samples** `(int)` – num samples in the dataset
- **warmup_epochs** `(int)` – epochs to warmup the lr for
- **lr** – the optimizer learning rate
- **opt_weight_decay** – the optimizer weight decay
- **loss_temperature** – the loss temperature

nt_xent_loss (*out_1*, *out_2*, *temperature*, *eps*=1e-06)

assume *out_1* and *out_2* are normalized *out_1*: [batch_size, dim] *out_2*: [batch_size, dim]

23.2.6 SwAV

PyTorch Lightning implementation of SwAV Adapted from the [official implementation](#)

Paper authors: Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, Armand Joulin.

Implementation adapted by:

- Ananya Harsh Jha

To Train:

```
import pytorch_lightning as pl
from pl_bolts.models.self_supervised import SwAV
from pl_bolts.datamodules import STL10DataModule
from pl_bolts.models.self_supervised.swav.transforms import (
    SwAVTrainDataTransform, SwAVEvalDataTransform
)
from pl_bolts.transforms.dataset_normalizations import stl10_normalization

# data
batch_size = 128
dm = STL10DataModule(data_dir='.', batch_size=batch_size)
dm.train_dataloader = dm.train_dataloader_mixed
dm.val_dataloader = dm.val_dataloader_mixed

dm.train_transforms = SwAVTrainDataTransform(
    normalize=stl10_normalization()
)
```

(continues on next page)

(continued from previous page)

```

dm.val_transforms = SwAVEvalDataTransform(
    normalize=stl10_normalization()
)

# model
model = SwAV(
    gpus=1,
    num_samples=dm.num_unlabeled_samples,
    dataset='stl10',
    batch_size=batch_size
)

# fit
trainer = pl.Trainer(precision=16)
trainer.fit(model)

```

Pre-trained ImageNet

We have included an option to directly load [ImageNet weights](#) provided by FAIR into bolts.

You can load the pretrained model using:

ImageNet pretrained model:

```

from pl_bolts.models.self_supervised import SwAV

weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/swav/swav_imagenet/
↳swav_imagenet.pth.tar'
swav = SwAV.load_from_checkpoint(weight_path, strict=True)

swav.freeze()

```

STL-10 baseline

The original paper does not provide baselines on STL10.

Table 4: STL-10 implementation results

Implementation	test acc	Encoder	Optimizer	Batch	Queue used	Epochs	Hardware	LR
Ours	86.72	SwAV resnet50	LARS	128	No	100 (~9 hr)	1 V100 (16GB)	1e-3

- [Pre-training tensorboard link](#)

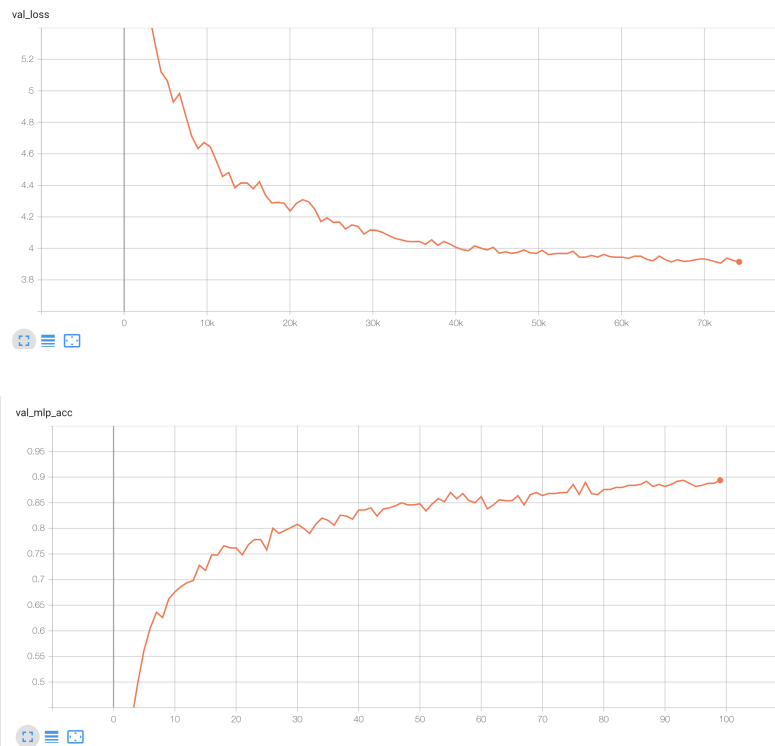
STL-10 pretrained model:

```
from pl_bolts.models.self_supervised import SwAV

weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/swav/checkpoints/
↪swav_stl10.pth.tar'
swav = SwAV.load_from_checkpoint(weight_path, strict=False)

swav.freeze()
```

Pre-training:

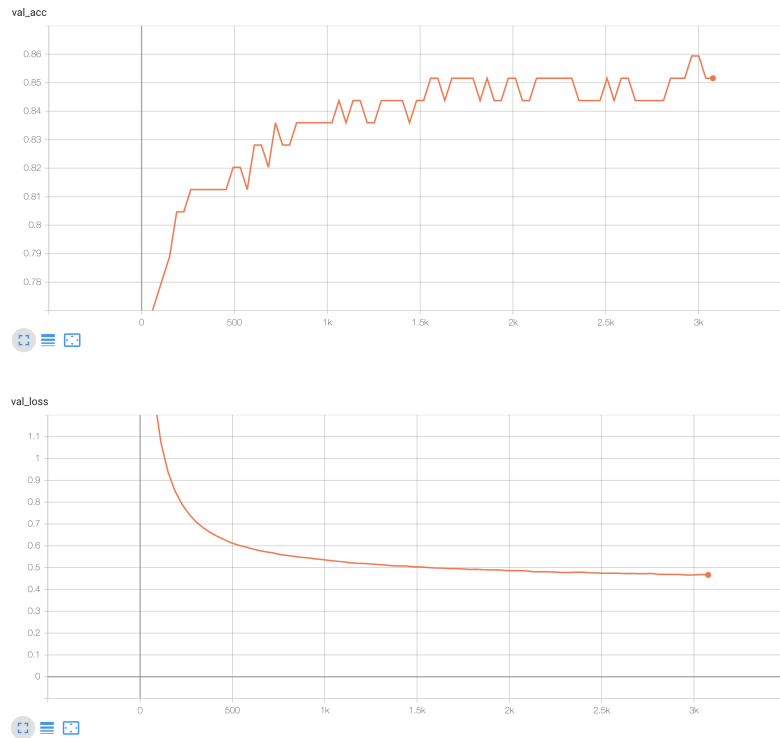


Fine-tuning (Single layer MLP, 1024 hidden units):

To reproduce:

```
# pretrain
python swav_module.py
```

(continues on next page)



(continued from previous page)

```

--online_ft
--gpus 1
--lars_wrapper
--batch_size 128
--learning_rate 1e-3
--gaussian_blur
--queue_length 0
--jitter_strength 1.
--nmb_prototypes 512

# finetune
python swav_finetuner.py
--gpus 8
--ckpt_path path/to/simclr/ckpt
--dataset imagenet
--data_dir path/to/imagenet/dataset
--batch_size 256
--num_workers 16
--learning_rate 0.8
--nesterov True
--num_epochs 90

```

Imagenet baseline for SwAV

Table 5: Cifar-10 implementation results

Implementation	test acc	Encoder	Optimizer	Batch	Epochs	Hardware	LR
Original	75.3	resnet50	LARS	4096	800	64 V100s	4.8
Ours	74	resnet50	LARS-SGD	4096	800	64 V100 (16GB)	4.8

Imagenet pretrained model:

```
from pl_bolts.models.self_supervised import SwAV

weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/swav/bolts_swav_
↪imagenet/swav_imagenet.ckpt'
swav = SwAV.load_from_checkpoint(weight_path, strict=False)

swav.freeze()
```

SwAV API

```
class pl_bolts.models.self_supervised.SwAV(gpus, num_samples, batch_size,
dataset, num_nodes=1,
arch='resnet50', hidden_mlp=2048,
feat_dim=128, warmup_epochs=10,
max_epochs=100, nmb_prototypes=3000,
freeze_prototypes_epochs=1, tem-
perature=0.1, sinkhorn_iterations=3,
queue_length=0, queue_path='queue',
epoch_queue_starts=15,
crops_for_assign=[0, 1], nmb_crops=[2,
6], first_conv=True, maxpool1=True,
optimizer='adam', lars_wrapper=True,
exclude_bn_bias=False, start_lr=0.0,
learning_rate=0.001, final_lr=0.0,
weight_decay=1e-06, epsilon=0.05,
**kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Parameters

- **gpus** `(int)` – number of gpus per node used in training, passed to SwAV module to manage the queue and select distributed sinkhorn
- **num_nodes** `(int)` – number of nodes to train on
- **num_samples** `(int)` – number of image samples used for training
- **batch_size** `(int)` – batch size per GPU in ddp

- **dataset** (str) – dataset being used for train/val
- **arch** (str) – encoder architecture used for pre-training
- **hidden_mlp** (int) – hidden layer of non-linear projection head, set to 0 to use a linear projection head
- **feat_dim** (int) – output dim of the projection head
- **warmup_epochs** (int) – apply linear warmup for this many epochs
- **max_epochs** (int) – epoch count for pre-training
- **nmb_prototypes** (int) – count of prototype vectors
- **freeze_prototypes_epochs** (int) – epoch till which gradients of prototype layer are frozen
- **temperature** (float) – loss temperature
- **sinkhorn_iterations** (int) – iterations for sinkhorn normalization
- **queue_length** (int) – set queue when batch size is small, must be divisible by total batch-size (i.e. total_gpus * batch_size), set to 0 to remove the queue
- **queue_path** (str) – folder within the logs directory
- **epoch_queue_starts** (int) – start using the queue after this epoch
- **crops_for_assign** (list) – list of crop ids for computing assignment
- **nmb_crops** (list) – number of global and local crops, ex: [2, 6]
- **first_conv** (bool) – keep first conv same as the original resnet architecture, if set to false it is replace by a kernel 3, stride 1 conv (cifar-10)
- **maxpool1** (bool) – keep first maxpool layer same as the original resnet architecture, if set to false, first maxpool is turned off (cifar10, maybe stl10)
- **optimizer** (str) – optimizer to use
- **lars_wrapper** (bool) – use LARS wrapper over the optimizer
- **exclude_bn_bias** (bool) – exclude batchnorm and bias layers from weight decay in optimizers
- **start_lr** (float) – starting lr for linear warmup
- **learning_rate** (float) – learning rate
- **final_lr** (float) – float = final learning rate for cosine weight decay
- **weight_decay** (float) – weight decay for optimizer
- **epsilon** (float) – epsilon val for swav assignments

LEARNING RATE SCHEDULERS

This package lists common learning rate schedulers across research domains (This is a work in progress. If you have any learning rate schedulers you want to contribute, please submit a PR!)

Note: this module is a work in progress

24.1 Your Learning Rate Scheduler

We're cleaning up many of our learning rate schedulers, but in the meantime, submit a PR to add yours here!

24.2 Linear Warmup Cosine Annealing Learning Rate Scheduler

```
class pl_bolts.optimizers.lr_scheduler.LinearWarmupCosineAnnealingLR(optimizer,
                                                                    warmup_epochs,
                                                                    max_epochs,
                                                                    warmup_start_lr=0.0,
                                                                    eta_min=0.0,
                                                                    last_epoch=-1)
```

Bases: torch.optim.lr_scheduler._LRScheduler

Sets the learning rate of each parameter group to follow a linear warmup schedule between warmup_start_lr and base_lr followed by a cosine annealing schedule between base_lr and eta_min.

Warning: It is recommended to call `step()` for `LinearWarmupCosineAnnealingLR` after each iteration as calling it after each epoch will keep the starting lr at warmup_start_lr for the first epoch which is 0 in most cases.

Warning: passing epoch to `step()` is being deprecated and comes with an EPOCH_DEPRECATION_WARNING. It calls the `_get_closed_form_lr()` method for this scheduler instead of `get_lr()`. Though this does not change the behavior of the scheduler, when passing epoch param to `step()`, the user should call the `step()` function before calling train and validation methods.

Example

```
>>> layer = nn.Linear(10, 1)
>>> optimizer = Adam(layer.parameters(), lr=0.02)
>>> scheduler = LinearWarmupCosineAnnealingLR(optimizer, warmup_epochs=10, max_
↳ epochs=40)
>>> #
>>> # the default case
>>> for epoch in range(40):
...     # train(...)
...     # validate(...)
...     scheduler.step()
>>> #
>>> # passing epoch param case
>>> for epoch in range(40):
...     scheduler.step(epoch)
...     # train(...)
...     # validate(...)
```

Parameters

- **optimizer** *(Optimizer)* – Wrapped optimizer.
- **warmup_epochs** *(int)* – Maximum number of iterations for linear warmup
- **max_epochs** *(int)* – Maximum number of iterations
- **warmup_start_lr** *(float)* – Learning rate to start the linear warmup. Default: 0.
- **eta_min** *(float)* – Minimum learning rate. Default: 0.
- **last_epoch** *(int)* – The index of last epoch. Default: -1.

get_lr()

Compute learning rate using chainable form of the scheduler

Return type `List[float]`

SELF-SUPERVISED LEARNING TRANSFORMS

These transforms are used in various self-supervised learning approaches.

25.1 CPC transforms

Transforms used for CPC

25.1.1 CIFAR-10 Train (c)

```
class pl_bolts.models.self_supervised.cpc.transforms.CPCTrainTransformsCIFAR10 (patch_size=8,
                                                                              over-
                                                                              lap=4)
```

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
img_jitter
col_jitter
rnd_gray
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=patch_size // 2)
```

Example:

```
# in a regular dataset
CIFAR10(..., transforms=CPCTrainTransformsCIFAR10())

# in a DataModule
module = CIFAR10DataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
→ transforms=CPCTrainTransformsCIFAR10())
```

Parameters

- **patch_size** – size of patches when cutting up the image into overlapping patches
- **overlap** – how much to overlap patches

`__call__(inp)`
Call self as a function.

25.1.2 CIFAR-10 Eval (c)

`class pl_bolts.models.self_supervised.cpc.transforms.CPCEvalTransformsCIFAR10` (*patch_size=8, over-lap=4*)

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=overlap)
```

Example:

```
# in a regular dataset
CIFAR10(..., transforms=CPCEvalTransformsCIFAR10())

# in a DataModule
module = CIFAR10DataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
    ↪ transforms=CPCEvalTransformsCIFAR10())
```

Parameters

- **patch_size** `(int)` – size of patches when cutting up the image into overlapping patches
- **overlap** `(int)` – how much to overlap patches

`__call__(inp)`
Call self as a function.

25.1.3 Imagenet Train (c)

`class pl_bolts.models.self_supervised.cpc.transforms.CPCTrainTransformsImageNet128` (*patch_size=128, over-lap=16*)

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=patch_size // 2)
```

Example:

```
# in a regular dataset
Imagenet(..., transforms=CPCTrainTransformsImageNet128())

# in a DataModule
module = ImagenetDataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
↳ transforms=CPCTrainTransformsImageNet128())
```

Parameters

- **patch_size** (int) – size of patches when cutting up the image into overlapping patches
- **overlap** (int) – how much to overlap patches

__call__(inp)

Call self as a function.

25.1.4 Imagenet Eval (c)

```
class pl_bolts.models.self_supervised.cpc.transforms.CPCEvalTransformsImageNet128(patch_size=
over-
lap=16)
```

Bases: object

Transforms used for CPC:

Transforms:

```
random_flip
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=patch_size // 2)
```

Example:

```
# in a regular dataset
Imagenet(..., transforms=CPCEvalTransformsImageNet128())

# in a DataModule
module = ImagenetDataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
↳ transforms=CPCEvalTransformsImageNet128())
```

Parameters

- **patch_size** (int) – size of patches when cutting up the image into overlapping patches
- **overlap** (int) – how much to overlap patches

__call__(inp)

Call self as a function.

25.1.5 STL-10 Train (c)

class `pl_bolts.models.self_supervised.cpc.transforms.CPCTrainTransformsSTL10` (*patch_size=16,*
over-
lap=8)

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
img_jitter
col_jitter
rnd_gray
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=patch_size // 2)
```

Example:

```
# in a regular dataset
STL10(..., transforms=CPCTrainTransformsSTL10())

# in a DataModule
module = STL10DataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
→ transforms=CPCTrainTransformsSTL10())
```

Parameters

- **patch_size** `(int)` – size of patches when cutting up the image into overlapping patches
- **overlap** `(int)` – how much to overlap patches

`__call__` (*inp*)

Call self as a function.

25.1.6 STL-10 Eval (c)

class `pl_bolts.models.self_supervised.cpc.transforms.CPCEvalTransformsSTL10` (*patch_size=16,*
over-
lap=8)

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=patch_size // 2)
```

Example:

```
# in a regular dataset
STL10(..., transforms=CPCEvalTransformsSTL10())

# in a DataModule
module = STL10DataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
↳ transforms=CPCEvalTransformsSTL10())
```

Parameters

- **patch_size** (int) – size of patches when cutting up the image into overlapping patches
- **overlap** (int) – how much to overlap patches

__call__(inp)

Call self as a function.

25.2 AMDIM transforms

Transforms used for AMDIM

25.2.1 CIFAR-10 Train (a)

class pl_bolts.models.self_supervised.amdim.transforms.**AMDIMTrainTransformsCIFAR10**

Bases: `object`

Transforms applied to AMDIM

Transforms:

```
img_jitter,
col_jitter,
rnd_gray,
transforms.ToTensor(),
normalize
```

Example:

```
x = torch.rand(5, 3, 32, 32)

transform = AMDIMTrainTransformsCIFAR10()
(view1, view2) = transform(x)
```

__call__(inp)

Call self as a function.

25.2.2 CIFAR-10 Eval (a)

class `pl_bolts.models.self_supervised.amdim.transforms.AMDIMEvalTransformsCIFAR10`

Bases: `object`

Transforms applied to AMDIM

Transforms:

```
transforms.ToTensor(),
normalize
```

Example:

```
x = torch.rand(5, 3, 32, 32)

transform = AMDIMEvalTransformsCIFAR10()
(view1, view2) = transform(x)
```

`__call__` (*inp*)

Call self as a function.

25.2.3 Imagenet Train (a)

class `pl_bolts.models.self_supervised.amdim.transforms.AMDIMTrainTransformsImageNet128` (*height*)

Bases: `object`

Transforms applied to AMDIM

Transforms:

```
img_jitter,
col_jitter,
rnd_gray,
transforms.ToTensor(),
normalize
```

Example:

```
x = torch.rand(5, 3, 128, 128)

transform = AMDIMTrainTransformsSTL10()
(view1, view2) = transform(x)
```

`__call__` (*inp*)

Call self as a function.

25.2.4 Imagenet Eval (a)

class `pl_bolts.models.self_supervised.amdim.transforms.AMDIMEvalTransformsImageNet128` (*height*)
 Bases: `object`

Transforms applied to AMDIM

Transforms:

```
transforms.Resize(height + 6, interpolation=3),
transforms.CenterCrop(height),
transforms.ToTensor(),
normalize
```

Example:

```
x = torch.rand(5, 3, 128, 128)

transform = AMDIMEvalTransformsImageNet128()
view1 = transform(x)
```

`__call__` (*inp*)
 Call self as a function.

25.2.5 STL-10 Train (a)

class `pl_bolts.models.self_supervised.amdim.transforms.AMDIMTrainTransformsSTL10` (*height=64*)
 Bases: `object`

Transforms applied to AMDIM

Transforms:

```
img_jitter,
col_jitter,
rnd_gray,
transforms.ToTensor(),
normalize
```

Example:

```
x = torch.rand(5, 3, 64, 64)

transform = AMDIMTrainTransformsSTL10()
(view1, view2) = transform(x)
```

`__call__` (*inp*)
 Call self as a function.

25.2.6 STL-10 Eval (a)

class `pl_bolts.models.self_supervised.amdim.transforms.AMDIMEvalTransformsSTL10` (*height=64*)
Bases: `object`

Transforms applied to AMDIM

Transforms:

```
transforms.Resize(height + 6, interpolation=3),
transforms.CenterCrop(height),
transforms.ToTensor(),
normalize
```

Example:

```
x = torch.rand(5, 3, 64, 64)

transform = AMDIMTrainTransformsSTL10()
view1 = transform(x)
```

`__call__` (*inp*)
Call self as a function.

25.3 MOCO V2 transforms

Transforms used for MOCO V2

25.3.1 CIFAR-10 Train (m2)

class `pl_bolts.models.self_supervised.moco.transforms.Moco2TrainCIFAR10Transforms` (*height=32*)
Bases: `object`

Moco 2 augmentation: <https://arxiv.org/pdf/2003.04297.pdf>

`__call__` (*inp*)
Call self as a function.

25.3.2 CIFAR-10 Eval (m2)

class `pl_bolts.models.self_supervised.moco.transforms.Moco2EvalCIFAR10Transforms` (*height=32*)
Bases: `object`

Moco 2 augmentation: <https://arxiv.org/pdf/2003.04297.pdf>

`__call__` (*inp*)
Call self as a function.

25.3.3 Imagenet Train (m2)

```
class pl_bolts.models.self_supervised.moco.transforms.Moco2TrainSTL10Transforms (height=64)
    Bases: object

    Moco 2 augmentation: https://arxiv.org/pdf/2003.04297.pdf

    __call__ (inp)
        Call self as a function.
```

25.3.4 Imagenet Eval (m2)

```
class pl_bolts.models.self_supervised.moco.transforms.Moco2EvalSTL10Transforms (height=64)
    Bases: object

    Moco 2 augmentation: https://arxiv.org/pdf/2003.04297.pdf

    __call__ (inp)
        Call self as a function.
```

25.3.5 STL-10 Train (m2)

```
class pl_bolts.models.self_supervised.moco.transforms.Moco2TrainImagenetTransforms (height=128)
    Bases: object

    Moco 2 augmentation: https://arxiv.org/pdf/2003.04297.pdf

    __call__ (inp)
        Call self as a function.
```

25.3.6 STL-10 Eval (m2)

```
class pl_bolts.models.self_supervised.moco.transforms.Moco2EvalImagenetTransforms (height=128)
    Bases: object

    Moco 2 augmentation: https://arxiv.org/pdf/2003.04297.pdf

    __call__ (inp)
        Call self as a function.
```

25.4 SimCLR transforms

Transforms used for SimCLR

25.4.1 Train (sc)

```
class pl_bolts.models.self_supervised.simclr.transforms.SimCLRTrainDataTransform(input_height=  
gaus-  
sian_blur=True  
jit-  
ter_strength=1.  
nor-  
mal-  
ize=None)
```

Bases: `object`

Transforms for SimCLR

Transform:

```
RandomResizedCrop(size=self.input_height)  
RandomHorizontalFlip()  
RandomApply([color_jitter], p=0.8)  
RandomGrayscale(p=0.2)  
GaussianBlur(kernel_size=int(0.1 * self.input_height))  
transforms.ToTensor()
```

Example:

```
from pl_bolts.models.self_supervised.simclr.transforms import_  
↳ SimCLRTrainDataTransform  
  
transform = SimCLRTrainDataTransform(input_height=32)  
x = sample()  
(xi, xj) = transform(x)
```

`__call__` (*sample*)
Call self as a function.

25.4.2 Eval (sc)

```
class pl_bolts.models.self_supervised.simclr.transforms.SimCLREvalDataTransform(input_height=2  
gaus-  
sian_blur=True  
jit-  
ter_strength=1.  
nor-  
mal-  
ize=None)
```

Bases: `pl_bolts.models.self_supervised.simclr.transforms.SimCLRTrainDataTransform`

Transforms for SimCLR

Transform:

```
Resize(input_height + 10, interpolation=3)  
transforms.CenterCrop(input_height),  
transforms.ToTensor()
```

Example:

```
from pl_bolts.models.self_supervised.simclr.transforms import   
↳ SimCLREvalDataTransform  
  
transform = SimCLREvalDataTransform(input_height=32)  
x = sample()  
(xi, xj) = transform(x)
```


SELF-SUPERVISED LEARNING

Collection of useful functions for self-supervised learning

26.1 Identity class

Example:

```
from pl_bolts.utils import Identity
```

```
class pl_bolts.utils.self_supervised.Identity
```

Bases: `torch.nn.Module`

An identity class to replace arbitrary layers in pretrained models

Example:

```
from pl_bolts.utils import Identity
```

```
model = resnet18()  
model.fc = Identity()
```

26.2 SSL-ready resnets

Torchvision resnets with the fc layers removed and with the ability to return all feature maps instead of just the last one.

Example:

```
from pl_bolts.utils.self_supervised import torchvision_ssl_encoder  
  
resnet = torchvision_ssl_encoder('resnet18', pretrained=False, return_all_feature_  
    ↪maps=True)  
x = torch.rand(3, 3, 32, 32)  
  
feat_maps = resnet(x)
```

```
pl_bolts.utils.self_supervised.torchvision_ssl_encoder(name, pretrained=False, re-  
    turn_all_feature_maps=False)
```

Return type `Module`

26.3 SSL backbone finetuner

```
class pl_bolts.models.self_supervised.ssl_finetuner.SSLFineTuner (backbone,
                                                                in_features=2048,
                                                                num_classes=1000,
                                                                epochs=100,
                                                                hid-
                                                                den_dim=None,
                                                                dropout=0.0,
                                                                learn-
                                                                ing_rate=0.1,
                                                                weight_decay=1e-
                                                                06,          nes-
                                                                terov=False,
                                                                sched-
                                                                uler_type='cosine',
                                                                de-
                                                                cay_epochs=[60,
                                                                80],
                                                                gamma=0.1,
                                                                final_lr=0.0)
```

Bases: `pytorch_lightning.LightningModule`

Finetunes a self-supervised learning backbone using the standard evaluation protocol of a singular layer MLP with 1024 units

Example:

```
from pl_bolts.utils.self_supervised import SSLFineTuner
from pl_bolts.models.self_supervised import CPCV2
from pl_bolts.datamodules import CIFAR10DataModule
from pl_bolts.models.self_supervised.cpc.transforms import _
↳CPCEvalTransformsCIFAR10,
↳CPCTrainTransformsCIFAR10

# pretrained model
backbone = CPCV2.load_from_checkpoint(PATH, strict=False)

# dataset + transforms
dm = CIFAR10DataModule(data_dir='.')
dm.train_transforms = CPCTrainTransformsCIFAR10()
dm.val_transforms = CPCEvalTransformsCIFAR10()

# finetuner
finetuner = SSLFineTuner(backbone, in_features=backbone.z_dim, num_
↳classes=backbone.num_classes)

# train
trainer = pl.Trainer()
trainer.fit(finetuner, dm)
```

(continues on next page)

(continued from previous page)

```
# test
trainer.test(datamodule=dm)
```

Parameters

- **backbone** (Module) – a pretrained model
- **in_features** (int) – feature dim of backbone outputs
- **num_classes** (int) – classes of the dataset
- **hidden_dim** (Optional[int]) – dim of the MLP (1024 default used in self-supervised literature)

SEMI-SUPERVISED LEARNING

Collection of utilities for semi-supervised learning where some part of the data is labeled and the other part is not.

27.1 Balanced classes

Example:

```
from pl_bolts.utils.semi_supervised import balance_classes
```

```
pl_bolts.utils.semi_supervised.balance_classes(X, Y, batch_size)
```

Makes sure each batch has an equal amount of data from each class. Perfect balance

Parameters

- **X** (Union[`Tensor`, `ndarray`]) – input features
- **Y** (Union[`Tensor`, `ndarray`, `Sequence[int]`]) – mixed labels (ints)
- **batch_size** (int) – the ultimate batch size

Return type `Tuple[ndarray, ndarray]`

27.2 half labeled batches

Example:

```
from pl_bolts.utils.semi_supervised import balance_classes
```

```
pl_bolts.utils.semi_supervised.generate_half_labeled_batches(smaller_set_X,  
                                                             smaller_set_Y,  
                                                             larger_set_X,  
                                                             larger_set_Y,  
                                                             batch_size)
```

Given a labeled dataset and an unlabeled dataset, this function generates a joint pair where half the batches are labeled and the other half is not

Return type `Tuple[ndarray, ndarray]`

SELF-SUPERVISED LEARNING CONTRASTIVE TASKS

This section implements popular contrastive learning tasks used in self-supervised learning.

28.1 FeatureMapContrastiveTask

This task compares sets of feature maps.

In general the feature map comparison pretext task uses triplets of features. Here are the abstract steps of comparison.

Generate multiple views of the same image

```
x1_view_1 = data_augmentation(x1)
x1_view_2 = data_augmentation(x1)
```

Use a different example to generate additional views (usually within the same batch or a pool of candidates)

```
x2_view_1 = data_augmentation(x2)
x2_view_2 = data_augmentation(x2)
```

Pick 3 views to compare, these are the anchor, positive and negative features

```
anchor = x1_view_1
positive = x1_view_2
negative = x2_view_1
```

Generate feature maps for each view

```
(a0, a1, a2) = encoder(anchor)
(p0, p1, p2) = encoder(positive)
```

Make a comparison for a set of feature maps

```
phi = some_score_function()

# the '01' comparison
score = phi(a0, p1)

# and can be bidirectional
score = phi(p0, a1)
```

In practice the contrastive task creates a $B \times B$ matrix where B is the batch size. The diagonals for set 1 of feature maps are the anchors, the diagonals of set 2 of the feature maps are the positives, the non-diagonals of set 1 are the negatives.

```
class pl_bolts.losses.self_supervised_learning.FeatureMapContrastiveTask (comparisons='00,
                                                                    11',
                                                                    tclip=10.0,
                                                                    bidi-
                                                                    rec-
                                                                    tional=True)
```

Bases: `torch.nn.Module`

Performs an anchor, positive negative pair comparison for each each tuple of feature maps passed.

```
# extract feature maps
pos_0, pos_1, pos_2 = encoder(x_pos)
anc_0, anc_1, anc_2 = encoder(x_anchor)

# compare only the 0th feature maps
task = FeatureMapContrastiveTask('00')
loss, regularizer = task((pos_0), (anc_0))

# compare (pos_0 to anc_1) and (pos_0, anc_2)
task = FeatureMapContrastiveTask('01, 02')
losses, regularizer = task((pos_0, pos_1, pos_2), (anc_0, anc_1, anc_2))
loss = losses.sum()

# compare (pos_1 vs a anc_random)
task = FeatureMapContrastiveTask('0r')
loss, regularizer = task((pos_0, pos_1, pos_2), (anc_0, anc_1, anc_2))
```

```
# with bidirectional the comparisons are done both ways
task = FeatureMapContrastiveTask('01, 02')

# will compare the following:
# 01: (pos_0, anc_1), (anc_0, pos_1)
# 02: (pos_0, anc_2), (anc_0, pos_2)
```

Parameters

- **comparisons** (str) – groupings of feature map indices to compare (zero indexed, ‘r’ means random) ex: ‘00, 1r’
- **tclip** (float) – stability clipping value
- **bidirectional** (bool) – if true, does the comparison both ways

forward (anchor_maps, positive_maps)

Takes in a set of tuples, each tuple has two feature maps with all matching dimensions

Example

```
>>> import torch
>>> from pytorch_lightning import seed_everything
>>> seed_everything(0)
0
>>> a1 = torch.rand(3, 5, 2, 2)
>>> a2 = torch.rand(3, 5, 2, 2)
>>> b1 = torch.rand(3, 5, 2, 2)
>>> b2 = torch.rand(3, 5, 2, 2)
...

```

(continues on next page)

(continued from previous page)

```
>>> task = FeatureMapContrastiveTask('01, 11')
...
>>> losses, regularizer = task((a1, a2), (b1, b2))
>>> losses
tensor([2.2351, 2.1902])
>>> regularizer
tensor(0.0324)
```

static parse_map_indexes (*comparisons*)

Example:

```
>>> FeatureMapContrastiveTask.parse_map_indexes('11')
[(1, 1)]
>>> FeatureMapContrastiveTask.parse_map_indexes('11,59')
[(1, 1), (5, 9)]
>>> FeatureMapContrastiveTask.parse_map_indexes('11,59, 2r')
[(1, 1), (5, 9), (2, -1)]
```

28.2 Context prediction tasks

The following tasks aim to predict a target using a context representation.

28.2.1 CPCTask

This is the predictive task from CPC (v2).

```
task = CPCTask(num_input_channels=32)

# (batch, channels, rows, cols)
# this should be thought of as 49 feature vectors, each with 32 dims
Z = torch.random.rand(3, 32, 7, 7)

loss = task(Z)
```

class pl_bolts.losses.self_supervised_learning.**CPCTask** (*num_input_channels*,
target_dim=64, em-
bed_scale=0.1)

Bases: `torch.nn.Module`

Loss used in CPC

CONTRIBUTING

Welcome to the PyTorch Lightning community! We're building the most advanced research platform on the planet to implement the latest, best practices that the amazing PyTorch team rolls out!

29.1 Bolts Design Principles

We encourage all sorts of contributions you're interested in adding! When coding for Bolts, please follow these principles.

29.1.1 Simple Internal Code

It's useful for users to look at the code and understand very quickly what's happening. Many users won't be engineers. Thus we need to value clear, simple code over condensed ninja moves. While that's super cool, this isn't the project for that :)

29.1.2 Force User Decisions To Best Practices

There are 1,000 ways to do something. However, eventually one popular solution becomes standard practice, and everyone follows. We try to find the best way to solve a particular problem, and then force our users to use it for readability and simplicity.

When something becomes a best practice, we add it to the framework. This is usually something like bits of code in utils or in the model file that everyone keeps adding over and over again across projects. When this happens, bring that code inside the trainer and add a flag for it.

29.1.3 Backward-compatible API

We all hate updating our deep learning packages because we don't want to refactor a bunch of stuff. In bolts, we make sure every change we make which could break an API is backward compatible with good deprecation warnings.

29.1.4 Gain User Trust

As a researcher, you can't have any part of your code going wrong. So, make thorough tests to ensure that every implementation of a new trick or subtle change is correct.

29.1.5 Interoperability

PyTorch Lightning Bolts is highly interoperable with PyTorch Lightning and PyTorch.

29.2 Contribution Types

We are always looking for help implementing new features or fixing bugs.

A lot of good work has already been done in project mechanics (requirements/base.txt, setup.py, pep8, badges, ci, etc...) so we're in a good state there thanks to all the early contributors (even pre-beta release)!

29.2.1 Bug Fixes:

1. If you find a bug please submit a GitHub issue.
 - Make sure the title explains the issue.
 - Describe your setup, what you are trying to do, expected vs. actual behaviour. Please add configs and code samples.
 - Add details on how to reproduce the issue - a minimal test case is always best, colab is also great. Note, that the sample code shall be minimal and if needed with publicly available data.
2. Try to fix it or recommend a solution. We highly recommend to use test-driven approach:
 - Convert your minimal code example to a unit/integration test with assert on expected results.
 - Start by debugging the issue... You can run just this particular test in your IDE and draft a fix.
 - Verify that your test case fails on the master branch and only passes with the fix applied.
3. Submit a PR!

Note, even if you do not find the solution, sending a PR with a test covering the issue is a valid contribution and we can help you or finish it with you :]

29.2.2 New Features:

1. Submit a GitHub issue - describe what is the motivation of such feature (adding the use case or an example is helpful).
2. Let's discuss to determine the feature scope.
3. Submit a PR! We recommend test driven approach to adding new features as well:
 - Write a test for the functionality you want to add.
 - Write the functional code until the test passes.
4. Add/update the relevant tests!
 - [This PR](#) is a good example for adding a new metric, and [this one](#) for a new logger.

29.2.3 New Models:

PyTorch Lightning Bolts makes several research models for ready usage. Following are general guidelines for adding new models.

1. Models which are standard baselines
2. Whose results are reproduced properly either by us or by authors.
3. Top models which are not SOTA but highly cited for production usage / for other uses. (E.g. Mobile BERT, MobileNets, FBNNets).
4. Do not reinvent the wheel, natively support torchvision, torchtext, torchaudio models.
5. Use open source licensed models.

Please raise an issue before adding a new model. Please let us know why the particular model is important for bolts. There are tons of models that keep coming. It is very difficult to support every model.

29.2.4 Test cases:

Want to keep Lightning Bolts healthy? Love seeing those green tests? So do we! How to we keep it that way? We write tests! We value tests contribution even more than new features.

Tests are written using `pytest`. Tests in PyTorch Lightning bolts train model on a datamodule. Datamodule is lightning abstraction of representing dataloader and dataset. Model is checked by simply calling `.fit()` function over the datamodule.

Along with these we have tests for losses, callbacks and transforms as well.

Have a look at sample tests [here](#).

After you have added the respective tests, you can run the tests locally with make script:

```
make test
```

Want to add a new test case and not sure how? [Talk to us!](#)

29.3 Note before submitting the PR, make sure you have run `make isort`.

29.4 Guidelines

For this section, we refer to read the [parent PL guidelines](#)

Reminder

All added or edited code shall be the own original work of the particular contributor. If you use some third-party implementation, all such blocks/functions/modules shall be properly referred and if possible also agreed by code's author. For example - This code is inspired from `http://...` In case you adding new dependencies, make sure that they are compatible with the actual PyTorch Lightning license (ie. dependencies should be *at least* as permissive as the PyTorch Lightning license).

29.4.1 Question & Answer

1. How can I help/contribute?

All help is extremely welcome - reporting bugs, fixing documentation, adding test cases, solving issues and preparing bug fixes. To solve some issues you can start with label [good first issue](#) or chose something close to your domain with label [help wanted](#). Before you start to implement anything check that the issue description that it is clear and self-assign the task to you (if it is not possible, just comment that you take it and we assign it to you...).

2. Is there a recommendation for branch names?

We do not rely on the name convention so far you are working with your own fork. Anyway it would be nice to follow this convention `<type>/<issue-id>_<short-name>` where the types are: `bugfix`, `feature`, `docs`, `tests`, ...

3. I have a model in other framework than PyTorch, how do I add it here?

Since PyTorch Lightning is written on top of PyTorch. We need models in PyTorch only. Also, we would need same or equivalent results with PyTorch Lightning after converting the models from other frameworks.

PL BOLTS GOVERNANCE | PERSONS OF INTEREST

30.1 Core Maintainers

- William Falcon ([williamFalcon](#)) (Lightning founder)
- Jirka Borovec ([Borda](#))
- Annika Brundyn ([annikabrundyn](#))
- Ananya Harsh Jha ([ananyahjha93](#))
- Teddy Koker ([teddykoker](#))
- Akihiro Nitta ([akihironitta](#))

CHANGELOG

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

31.1 [0.3.0] - 2021-01-20

31.1.1 [0.3.0] - Added

- Added `input_channels` argument to UNet (#297)
- Added SwAV (#239, #348, #323)
- Added data monitor callbacks `ModuleDataMonitor` and `TrainingDataMonitor` (#285)
- Added DCGAN module (#403)
- Added `VisionDataModule` as parent class for `BinaryMNISTDataModule`, `CIFAR10DataModule`, `FashionMNISTDataModule`, and `MNISTDataModule` (#400)
- Added GIoU loss (#347)
- Added IoU loss (#469)
- Added semantic segmentation model `SemSegment` with UNet backend (#259)
- Added option to normalize latent interpolation images (#438)
- Added flags to datamodules (#388)
- Added metric GIoU (#347)
- Added Intersection over Union Metric/Loss (#469)
- Added SimSiam model (#407)
- Added gradient verification callback (#465)
- Added Backbones to FRCNN (#475)

31.1.2 [0.3.0] - Changed

- Decoupled datamodules from models (#332, #270)
- Set PyTorch Lightning 1.0 as the minimum requirement (#274)
- Moved `pl_bolts.callbacks.self_supervised.BYOLMAWeightUpdate` to `pl_bolts.callbacks.byol_updates.BYOLMAWeightUpdate` (#288)
- Moved `pl_bolts.callbacks.self_supervised.SSLOnlineEvaluator` to `pl_bolts.callbacks.ssl_online.SSLOnlineEvaluator` (#288)
- Moved `pl_bolts.datamodules.*_dataset` to `pl_bolts.datasets.*_dataset` (#275)
- Ensured sync across val/test step when using DDP (#371)
- Refactored CLI arguments of models (#394)
- Upgraded DQN to use `.log` (#404)
- Decoupled DataModules from models - CPCV2 (#386)
- Refactored datamodules/datasets (#338)
- Refactored Vision DataModules (#400)
- Refactored `pl_bolts.callbacks` (#477)
- Refactored the rest of `pl_bolts.models.self_supervised` (#481, #479)
- Update `[torchvision.utils.make_grid](https://pytorch.org/docs/stable/torchvision/utils.html#torchvision.utils.make_grid)` to `TensorboardGenerativeModelImageSampler` (#494)

31.1.3 [0.3.0] - Fixed

- Fixed duplicate warnings when optional packages are unavailable (#341)
- Fixed `ModuleNotFoundError` when importing datamodules (#303)
- Fixed cyclic imports in `pl_bolts.utils.self_supervised` (#350)
- Fixed VAE loss to use KL term of ELBO (#330)
- Fixed dataloaders of `MNISTDataModule` to use `self.batch_size` (#331)
- Fixed missing outputs in SSL hooks for PyTorch Lightning 1.0 (#277)
- Fixed `stl10` datamodule (#369)
- Fixes SimCLR transforms (#329)
- Fixed binary MNIST datamodule (#377)
- Fixed the end of batch size mismatch (#389)
- Fixed `batch_size` parameter for DataModules remaining (#344)
- Fixed CIFAR `num_samples` (#432)
- Fixed DQN `run_n_episodes` using the wrong environment variable (#525)

31.2 [0.2.5] - 2020-10-12

- Enabled PyTorch Lightning 1.0 compatibility

31.3 [0.2.4] - 2020-10-12

- Enabled manual returns ([#267](#))

31.4 [0.2.3] - 2020-10-12

31.4.1 [0.2.3] - Added

- Enabled PyTorch Lightning 0.10 compatibility ([#264](#))
- Added dummy datasets ([#266](#))
- Added `KittiDataModule` ([#248](#))
- Added UNet ([#247](#))
- Added reinforcement learning models, losses and datamodules ([#257](#))

31.5 [0.2.2] - 2020-09-14

- Fixed confused logit ([#222](#))

31.6 [0.2.1] - 2020-09-13

31.6.1 [0.2.1] - Added

- Added pretrained VAE with resnet encoders and decoders
- Added pretrained AE with resnet encoders and decoders
- Added CPC pretrained on CIFAR10 and STL10
- Verified BYOL implementation

31.6.2 [0.2.1] - Changed

- Dropped all dependencies except PyTorch Lightning and PyTorch
- Decoupled datamodules from GAN ([#206](#))
- Modularize AE & VAE ([#196](#))

31.6.3 [0.2.1] - Fixed

- Fixed gym (#221)
- Fix L1/L2 regularization (#216)
- Fix max_depth recursion crash in AsynchronousLoader (#191)

31.7 [0.2.0] - 2020-09-10

31.7.1 [0.2.0] - Added

- Enabled Apache License, Version 2.0

31.7.2 [0.2.0] - Changed

- Moved unnecessary dependencies to `__main__` section in BYOL (#176)

31.7.3 [0.2.0] - Fixed

- Fixed CPC STL10 finetune (#173)

31.8 [0.1.1] - 2020-08-23

31.8.1 [0.1.1] - Added

- Added Faster RCNN + Pscal VOC DataModule (#157)
- Added a better lars scheduling LARSWrapper (#162)
- Added CPC finetuner (#158)
- Added BinaryMNISTDataModule (#153)
- Added learning rate scheduler to BYOL (#148)
- Added Cityscapes DataModule (#136)
- Added learning rate scheduler LinearWarmupCosineAnnealingLR (#138)
- Added BYOL (#144)
- Added ConfusedLogitCallback (#118)
- Added an asynchronous single GPU dataloader. (#1521)

31.8.2 [0.1.1] - Fixed

- Fixed simclr finetuner (#165)
- Fixed STL10 finetuner (#164)
- Fixed Image GPT (#108)
- Fixed unused MNIST transforms in tran/val/test (#109)

31.8.3 [0.1.1] - Changed

- Enhanced train batch function (#107)

31.9 [0.1.0] - 2020-07-02

31.9.1 [0.1.0] - Added

- Added setup and repo structure
- Added requirements
- Added docs
- Added Manifest
- Added coverage
- Added MNIST template
- Added VAE template
- Added GAN + AE + MNIST
- Added Linear Regression
- Added Moco2g
- Added simclr
- Added RL module
- Added Loggers
- Added Transforms
- Added Tiny Datasets
- Added regularization to linear + logistic models
- Added Linear and Logistic Regression tests
- Added Image GPT
- Added Recommenders module

31.9.2 [0.1.0] - Changed

- Device is no longer set in the DQN model init
- Moved RL loss function to the losses module
- Moved `rl.common.experience` to `datamodules`
- `train_batch` function to VPG model to generate batch of data at each step (POC)
- Experience source no longer gets initialized with a device, instead the device is passed at each step()
- Refactored `ExperienceSource` classes to be handle multiple environments.

31.9.3 [0.1.0] - Removed

- Removed N-Step DQN as the latest version of the DQN supports N-Step by setting the `n_step` arg to `n`
- Deprecated `common.experience`

31.9.4 [0.1.0] - Fixed

- Documentation
- Doct tests
- CI pipeline
- Imports and pkg
- CPC fixes

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

32.1 `pl_bolts.callbacks` package

Collection of PyTorchLightning callbacks

```
class pl_bolts.callbacks.BatchGradientVerificationCallback(input_mapping=None,  
                                                         out-  
                                                         put_mapping=None,  
                                                         sample_idx=0,  
                                                         **kwargs)
```

Bases: `pl_bolts.callbacks.verification.base.VerificationCallbackBase`

The callback version of the `BatchGradientVerification` test. Verification is performed right before training begins.

Parameters

- **`input_mapping`** (`Optional[Callable]`) – An optional input mapping that returns all batched tensors in a input collection. See `BatchGradientVerification.check()` for more information.
- **`output_mapping`** (`Optional[Callable]`) – An optional output mapping that combines all batched tensors in the output collection into one big batch. See `BatchGradientVerification.check()` for more information.
- **`sample_idx`** (`int`) – The index of the batch sample to run the test for. See `BatchGradientVerification.check()` for more information.
- **`**kwargs`** – Additional arguments for the base class `VerificationCallbackBase`

`message` (**args, **kwargs*)

The message to be printed when the model does not pass the verification. If the message for warning and error differ, override the `warning_message()` and `error_message()` methods directly.

Parameters

- **`*args`** – Any positional arguments that are needed to construct the message.
- **`**kwargs`** – Any keyword arguments that are needed to construct the message.

Return type `str`

Returns The message as a string.

on_train_start (*trainer, pl_module*)

Return type None

class `pl_bolts.callbacks.BYOLMAWeightUpdate` (*initial_tau=0.996*)

Bases: `pytorch_lightning.Callback`

Weight update rule from BYOL.

Your model should have:

- `self.online_network`
- `self.target_network`

Updates the `target_network` params using an exponential moving average update rule weighted by `tau`. BYOL claims this keeps the `online_network` from collapsing.

Note: Automatically increases `tau` from `initial_tau` to 1.0 with every training step

Example:

```
# model must have 2 attributes
model = Model()
model.online_network = ...
model.target_network = ...

trainer = Trainer(callbacks=[BYOLMAWeightUpdate()])
```

Parameters `initial_tau` (`float`) – starting `tau`. Auto-updates with every training step

on_train_batch_end (*trainer, pl_module, outputs, batch, batch_idx, dataloader_idx*)

Return type None

update_tau (*pl_module, trainer*)

Return type `float`

update_weights (*online_net, target_net*)

Return type None

class `pl_bolts.callbacks.ModuleDataMonitor` (*submodules=None, log_every_n_steps=None*)

Bases: `pl_bolts.callbacks.data_monitor.DataMonitorBase`

Parameters

- **submodules** (`Union[bool, List[str], None]`) – If `True`, logs the in- and output histograms of every submodule in the `LightningModule`, including the root module itself. This parameter can also take a list of names of specific submodules (see example below). Default: `None`, logs only the in- and output of the root module.
- **log_every_n_steps** (`Optional[int]`) – The interval at which histograms should be logged. This defaults to the interval defined in the `Trainer`. Use this to override the `Trainer` default.

Note: A too low value for `log_every_n_steps` may have a significant performance impact especially when many submodules are involved, since the logging occurs during the forward pass. It should only be used for debugging purposes.

Example

```
# log the in- and output histograms of the `forward` in LightningModule
trainer = Trainer(callbacks=[ModuleDataMonitor()])

# all submodules in LightningModule
trainer = Trainer(callbacks=[ModuleDataMonitor(submodules=True)])

# specific submodules
trainer = Trainer(callbacks=[ModuleDataMonitor(submodules=["generator",
↪ "generator.conv1"])])
```

`_get_submodule_names` (*root_module*)

Return type `List[str]`

`_register_hook` (*module_name*, *module*)

Return type `RemovableHandle`

`on_train_end` (*trainer*, *pl_module*)

Return type `None`

`on_train_start` (*trainer*, *pl_module*)

Return type `None`

`GROUP_NAME_INPUT` = 'input'

`GROUP_NAME_OUTPUT` = 'output'

class `pl_bolts.callbacks.TrainingDataMonitor` (*log_every_n_steps=None*)

Bases: `pl_bolts.callbacks.data_monitor.DataMonitorBase`

Callback that logs the histogram of values in the batched data passed to *training_step*.

Parameters `log_every_n_steps` (`Optional[int]`) – The interval at which histograms should be logged. This defaults to the interval defined in the Trainer. Use this to override the Trainer default.

Example

```
# log histogram of training data passed to `LightningModule.training_step`
trainer = Trainer(callbacks=[TrainingDataMonitor()])
```

`on_train_batch_start` (*trainer*, *pl_module*, *batch*, **args*, ***kwargs*)

Return type `None`

`GROUP_NAME` = 'training_step'

```
class pl_bolts.callbacks.PrintTableMetricsCallback
```

Bases: `pytorch_lightning.callbacks.Callback`

Prints a table with the metrics in columns on every epoch end

Example:

```
from pl_bolts.callbacks import PrintTableMetricsCallback

callback = PrintTableMetricsCallback()
```

Pass into trainer like so:

```
trainer = pl.Trainer(callbacks=[callback])
trainer.fit(...)

# -----
# at the end of every epoch it will print
# -----

# loss|train_loss|val_loss|epoch
# -----
# 2.2541470527648926|2.2541470527648926|2.2158432006835938|0
```

```
on_epoch_end(trainer, pl_module)
```

Return type `None`

```
class pl_bolts.callbacks.SSLOnlineEvaluator(dataset, drop_p=0.2, hidden_dim=None,
                                           z_dim=None, num_classes=None)
```

Bases: `pytorch_lightning.Callback`

Attaches a MLP for fine-tuning using the standard self-supervised protocol.

Example:

```
# your model must have 2 attributes
model = Model()
model.z_dim = ... # the representation dim
model.num_classes = ... # the num of classes in the model

online_eval = SSLOnlineEvaluator(
    z_dim=model.z_dim,
    num_classes=model.num_classes,
    dataset='imagenet'
)
```

Parameters

- **dataset** (`str`) – if stl10, need to get the labeled batch
- **drop_p** (`float`) – Dropout probability
- **hidden_dim** (`Optional[int]`) – Hidden dimension for the fine-tune MLP
- **z_dim** (`Optional[int]`) – Representation dimension
- **num_classes** (`Optional[int]`) – Number of classes

```
get_representations(pl_module, x)
```

Return type `Tensor`

on_pretrainRoutine_start (*trainer, pl_module*)

Return type None

on_train_batch_end (*trainer, pl_module, outputs, batch, batch_idx, dataloader_idx*)

Return type None

on_validation_batch_end (*trainer, pl_module, outputs, batch, batch_idx, dataloader_idx*)

Return type None

to_device (*batch, device*)

Return type Tuple[[Tensor](#), [Tensor](#)]

```
class pl_bolts.callbacks.LatentDimInterpolator (interpolate_epoch_interval=20,  
                                              range_start=-5, range_end=5,  
                                              steps=11, num_samples=2, normal-  
                                              ize=True)
```

Bases: [pytorch_lightning.callbacks.Callback](#)

Interpolates the latent space for a model by setting all dims to zero and stepping through the first two dims increasing one unit at a time.

Default interpolates between [-5, 5] (-5, -4, -3, ..., 3, 4, 5)

Example:

```
from pl_bolts.callbacks import LatentDimInterpolator  
  
Trainer(callbacks=[LatentDimInterpolator()])
```

Parameters

- **interpolate_epoch_interval** ([int](#)) – default 20
- **range_start** ([int](#)) – default -5
- **range_end** ([int](#)) – default 5
- **steps** ([int](#)) – number of step between start and end
- **num_samples** ([int](#)) – default 2
- **normalize** ([bool](#)) – default True (change image to (0, 1) range)

interpolate_latent_space (*pl_module, latent_dim*)

Return type List[[Tensor](#)]

on_epoch_end (*trainer, pl_module*)

Return type None

```
class pl_bolts.callbacks.ConfusedLogitCallback (top_k, projection_factor=3,  
                                              min_logit_value=5.0, log-  
                                              ging_batch_interval=20,  
                                              max_logit_difference=0.1)
```

Bases: [pytorch_lightning.Callback](#)

Takes the logit predictions of a model and when the probabilities of two classes are very close, the model doesn't have high certainty that it should pick one vs the other class.

This callback shows how the input would have to change to swing the model from one label prediction to the other.

In this case, the network predicts a 5... but gives almost equal probability to an 8. The images show what about the original 5 would have to change to make it more like a 5 or more like an 8.

For each confused logit the confused images are generated by taking the gradient from a logit wrt an input for the top two closest logits.

Example:

```
from pl_bolts.callbacks.vision import ConfusedLogitCallback
trainer = Trainer(callbacks=[ConfusedLogitCallback()])
```

Note: Whenever called, this model will look for `self.last_batch` and `self.last_logits` in the `LightningModule`.

Note: This callback supports tensorboard only right now.

Authored by:

- Alfredo Canziani

Parameters

- **top_k** (*int*) – How many “offending” images we should plot
- **projection_factor** (*int*) – How much to multiply the input image to make it look more like this logit label
- **min_logit_value** (*float*) – Only consider logit values above this threshold
- **logging_batch_interval** (*int*) – How frequently to inspect/potentially plot something
- **max_logit_difference** (*float*) – When the top 2 logits are within this threshold we consider them confused

static `_ConfusedLogitCallback__draw_sample` (*fig, axarr, row_idx, col_idx, img, title*)

Return type `None`

`_plot` (*confusing_x, confusing_y, trainer, model, mask_idx*s)

Return type `None`

on_train_batch_end (*trainer, pl_module, outputs, batch, batch_idx, dataloader_idx*)

Return type `None`

```
class pl_bolts.callbacks.TensorboardGenerativeModelImageSampler(num_samples=3,
                                                                nrow=8,
                                                                padding=2,
                                                                normalize=False,
                                                                norm_range=None,
                                                                scale_each=False,
                                                                pad_value=0)
```

Bases: `pytorch_lightning.Callback`

Generates images and logs to tensorboard. Your model must implement the `forward` function for generation

Requirements:

```
# model must have img_dim arg
model.img_dim = (1, 28, 28)

# model forward must work for sampling
z = torch.rand(batch_size, latent_dim)
img_samples = your_model(z)
```

Example:

```
from pl_bolts.callbacks import TensorboardGenerativeModelImageSampler

trainer = Trainer(callbacks=[TensorboardGenerativeModelImageSampler()])
```

Parameters

- **num_samples** (`int`) – Number of images displayed in the grid. Default: 3.
- **nrow** (`int`) – Number of images displayed in each row of the grid. The final grid size is $(B / \text{nrow}, \text{nrow})$. Default: 8.
- **padding** (`int`) – Amount of padding. Default: 2.
- **normalize** (`bool`) – If `True`, shift the image to the range (0, 1), by the min and max values specified by range. Default: `False`.
- **norm_range** (`Optional[Tuple[int, int]]`) – Tuple (min, max) where min and max are numbers, then these numbers are used to normalize the image. By default, min and max are computed from the tensor.
- **scale_each** (`bool`) – If `True`, scale each image in the batch of images separately rather than the (min, max) over all images. Default: `False`.
- **pad_value** (`int`) – Value for the padded pixels. Default: 0.

on_epoch_end (*trainer, pl_module*)

Return type `None`

32.1.1 Subpackages

pl_bolts.callbacks.verification package

Submodules

pl_bolts.callbacks.verification.base module

class `pl_bolts.callbacks.verification.base.VerificationBase` (*model*)

Bases: `object`

Base class for model verification. All verifications should run with any `torch.nn.Module` unless otherwise stated.

Parameters **model** (`Module`) – The model to run verification for.

`_get_input_array_copy` (*input_array=None*)

Returns a deep copy of the example input array in cases where it is expected that the input changes during the verification process.

Parameters **`input_array`** (`Optional[Any]`) – The input to clone.

Return type `Any`

`_model_forward` (*input_array*)

Feeds the input array to the model via the `__call__` method.

Parameters **`input_array`** (`Any`) – The input that goes into the model. If it is a tuple, it gets interpreted as the sequence of positional arguments and is passed in by tuple unpacking. If it is a dict, the contents get passed in as named parameters by unpacking the dict. Otherwise, the input array gets passed in as a single argument.

Return type `Any`

Returns The output of the model.

`abstract check` (**args, **kwargs*)

Runs the actual test on the model. All verification classes must implement this.

Parameters

- **`*args`** – Any positional arguments that are needed to run the test
- **`*kwargs`** – Keyword arguments that are needed to run the test

Return type `bool`

Returns *True* if the test passes, and *False* otherwise. Some verifications can only be performed with a heuristic accuracy, thus the return value may not always reflect the true state of the system in these cases.

```
class pl_bolts.callbacks.verification.base.VerificationCallbackBase (warn=True,  
                                                                    er-  
                                                                    ror=False)
```

Bases: `pytorch_lightning.Callback`

Base class for model verification in form of a callback. This type of verification is expected to only work with `LightningModule` and will take the input array from `example_input_array` if needed.

Parameters

- **`warn`** (`bool`) – If *True*, prints a warning message when verification fails. Default: *True*.
- **`error`** (`bool`) – If *True*, prints an error message when verification fails. Default: *False*.

`_raise` (**args, **kwargs*)

Return type `None`

`error_message` (**args, **kwargs*)

The error message printed when the model does not pass the verification.

Return type `str`

`message` (**args, **kwargs*)

The message to be printed when the model does not pass the verification. If the message for warning and error differ, override the `warning_message()` and `error_message()` methods directly.

Parameters

- **`*args`** – Any positional arguments that are needed to construct the message.
- **`**kwargs`** – Any keyword arguments that are needed to construct the message.

Return type `str`

Returns The message as a string.

warning_message (**args, **kwargs*)

The warning message printed when the model does not pass the verification.

Return type `str`

`pl_bolts.callbacks.verification.batch_gradient` module

class `pl_bolts.callbacks.verification.batch_gradient.BatchGradientVerification` (*model*)

Bases: `pl_bolts.callbacks.verification.base.VerificationBase`

Checks if a model mixes data across the batch dimension. This can happen if reshape- and/or permutation operations are carried out in the wrong order or on the wrong tensor dimensions.

Parameters **model** (`Module`) – The model to run verification for.

check (*input_array, input_mapping=None, output_mapping=None, sample_idx=0*)

Runs the test for data mixing across the batch.

Parameters

- **input_array** (*Any*) – A dummy input for the model. Can be a tuple or dict in case the model takes multiple positional or named arguments.
- **input_mapping** (*Optional[Callable]*) – An optional input mapping that returns all batched tensors in a input collection. By default, we handle nested collections (tuples, lists, dicts) of tensors and pull them out. If your batch is a custom object, you need to provide this input mapping yourself. See `default_input_mapping()` for more information on the default behavior.
- **output_mapping** (*Optional[Callable]*) – An optional output mapping that combines all batched tensors in the output collection into one big batch of shape (B, N), where N is the total number of dimensions that follow the batch dimension in each tensor. By default, we handle nested collections (tuples, lists, dicts) of tensors and combine them automatically. See `default_output_mapping()` for more information on the default behavior.
- **sample_idx** (*int*) – The index *i* of the batch sample to run the test for. When computing the gradient of a loss value on the *i*-th output w.r.t. the whole input, we expect the gradient to be non-zero only on the *i*-th input sample and zero gradient on the rest of the batch.

Return type `bool`

Returns `True` if the data in the batch does not mix during the forward pass, and `False` otherwise.

class `pl_bolts.callbacks.verification.batch_gradient.BatchGradientVerificationCallback` (*input*

out-
put_
put_
sam-
ple_i
***kw*

Bases: `pl_bolts.callbacks.verification.base.VerificationCallbackBase`

The callback version of the `BatchGradientVerification` test. Verification is performed right before training begins.

Parameters

- **input_mapping** (`Optional[Callable]`) – An optional input mapping that returns all batched tensors in a input collection. See `BatchGradientVerification.check()` for more information.
- **output_mapping** (`Optional[Callable]`) – An optional output mapping that combines all batched tensors in the output collection into one big batch. See `BatchGradientVerification.check()` for more information.
- **sample_idx** (`int`) – The index of the batch sample to run the test for. See `BatchGradientVerification.check()` for more information.
- ****kwargs** – Additional arguments for the base class `VerificationCallbackBase`

message (`*args, **kwargs`)

The message to be printed when the model does not pass the verification. If the message for warning and error differ, override the `warning_message()` and `error_message()` methods directly.

Parameters

- ***args** – Any positional arguments that are needed to construct the message.
- ****kwargs** – Any keyword arguments that are needed to construct the message.

Return type `str`

Returns The message as a string.

on_train_start (`trainer, pl_module`)

Return type `None`

`pl_bolts.callbacks.verification.batch_gradient.collect_tensors` (`data`)

Filters all tensors in a collection and returns them in a list.

Return type `List[Tensor]`

`pl_bolts.callbacks.verification.batch_gradient.default_input_mapping` (`data`)

Finds all tensors in a (nested) collection that have the same batch size.

Parameters **data** (`Any`) – a tensor or a collection of tensors (tuple, list, dict, etc.).

Return type `List[Tensor]`

Returns A list of all tensors with the same batch dimensions. If the input was already a tensor, a one- element list with the tensor is returned.

```
>>> data = (torch.zeros(3, 1), "foo", torch.ones(3, 2), torch.rand(2))
>>> result = default_input_mapping(data)
>>> len(result)
2
>>> result[0].shape
torch.Size([3, 1])
>>> result[1].shape
torch.Size([3, 2])
```

`pl_bolts.callbacks.verification.batch_gradient.default_output_mapping` (`data`)

Pulls out all tensors in a output collection and combines them into one big batch for verification.

Parameters **data** (`Any`) – a tensor or a (nested) collection of tensors (tuple, list, dict, etc.).

Return type `Tensor`

Returns A float tensor with shape (B, N) where B is the batch size and N is the sum of (flattened) dimensions of all tensors in the collection. If the input was already a tensor, the tensor itself is returned.

Example

```
>>> data = (torch.rand(3, 5), "foo", torch.rand(3, 2, 4))
>>> result = default_output_mapping(data)
>>> result.shape
torch.Size([3, 13])
>>> data = {"one": torch.rand(3, 5), "two": torch.rand(3, 2, 1)}
>>> result = default_output_mapping(data)
>>> result.shape
torch.Size([3, 7])
```

pl_bolts.callbacks.vision package

Submodules

pl_bolts.callbacks.vision.confused_logit module

```
class pl_bolts.callbacks.vision.confused_logit.ConfusedLogitCallback(top_k,
                                                                    projec-
                                                                    tion_factor=3,
                                                                    min_logit_value=5.0,
                                                                    log-
                                                                    ging_batch_interval=20,
                                                                    max_logit_difference=0.1)
```

Bases: `pytorch_lightning.Callback`

Takes the logit predictions of a model and when the probabilities of two classes are very close, the model doesn't have high certainty that it should pick one vs the other class.

This callback shows how the input would have to change to swing the model from one label prediction to the other.

In this case, the network predicts a 5... but gives almost equal probability to an 8. The images show what about the original 5 would have to change to make it more like a 5 or more like an 8.

For each confused logit the confused images are generated by taking the gradient from a logit wrt an input for the top two closest logits.

Example:

```
from pl_bolts.callbacks.vision import ConfusedLogitCallback
trainer = Trainer(callbacks=[ConfusedLogitCallback()])
```

Note: Whenever called, this model will look for `self.last_batch` and `self.last_logits` in the `LightningModule`.

Note: This callback supports tensorboard only right now.

Authored by:

- Alfredo Canziani

Parameters

- **top_k** (*int*) – How many “offending” images we should plot
- **projection_factor** (*int*) – How much to multiply the input image to make it look more like this logit label
- **min_logit_value** (*float*) – Only consider logit values above this threshold
- **logging_batch_interval** (*int*) – How frequently to inspect/potentially plot something
- **max_logit_difference** (*float*) – When the top 2 logits are within this threshold we consider them confused

static `_ConfusedLogitCallback__draw_sample` (*fig, axarr, row_idx, col_idx, img, title*)

Return type `None`

`_plot` (*confusing_x, confusing_y, trainer, model, mask_idx*)

Return type `None`

`on_train_batch_end` (*trainer, pl_module, outputs, batch, batch_idx, dataloader_idx*)

Return type `None`

`pl_bolts.callbacks.vision.image_generation` module

class `pl_bolts.callbacks.vision.image_generation.TensorboardGenerativeModelImageSampler` (*num*)

Bases: `pytorch_lightning.Callback`

Generates images and logs to tensorboard. Your model must implement the `forward` function for generation

Requirements:

```
# model must have img_dim arg
model.img_dim = (1, 28, 28)

# model forward must work for sampling
z = torch.rand(batch_size, latent_dim)
img_samples = your_model(z)
```

Example:

```
from pl_bolts.callbacks import TensorboardGenerativeModelImageSampler

trainer = Trainer(callbacks=[TensorboardGenerativeModelImageSampler()])
```


Parameters

- **num_samples** (`int`) – Number of images displayed in the grid. Default: 3.
- **nrow** (`int`) – Number of images displayed in each row of the grid. The final grid size is $(B // nrow, nrow)$. Default: 8.
- **padding** (`int`) – Amount of padding. Default: 2.
- **normalize** (`bool`) – If `True`, shift the image to the range (0, 1), by the min and max values specified by `range`. Default: `False`.
- **norm_range** (`Optional[Tuple[int, int]]`) – Tuple (min, max) where min and max are numbers, then these numbers are used to normalize the image. By default, min and max are computed from the tensor.
- **scale_each** (`bool`) – If `True`, scale each image in the batch of images separately rather than the (min, max) over all images. Default: `False`.
- **pad_value** (`int`) – Value for the padded pixels. Default: 0.

`on_epoch_end` (*trainer, pl_module*)

Return type `None`

32.1.2 Submodules

`pl_bolts.callbacks.byol_updates` module

class `pl_bolts.callbacks.byol_updates.BYOLMAWeightUpdate` (*initial_tau=0.996*)

Bases: `pytorch_lightning.Callback`

Weight update rule from BYOL.

Your model should have:

- `self.online_network`
- `self.target_network`

Updates the `target_network` params using an exponential moving average update rule weighted by `tau`. BYOL claims this keeps the `online_network` from collapsing.

Note: Automatically increases `tau` from `initial_tau` to 1.0 with every training step

Example:

```
# model must have 2 attributes
model = Model()
model.online_network = ...
model.target_network = ...

trainer = Trainer(callbacks=[BYOLMAWeightUpdate()])
```

Parameters `initial_tau` (`float`) – starting `tau`. Auto-updates with every training step

`on_train_batch_end` (*trainer, pl_module, outputs, batch, batch_idx, dataloader_idx*)

Return type `None`

update_tau (*pl_module, trainer*)

Return type `float`

update_weights (*online_net, target_net*)

Return type `None`

pl_bolts.callbacks.data_monitor module

class `pl_bolts.callbacks.data_monitor.DataMonitorBase` (*log_every_n_steps=None*)

Bases: `pytorch_lightning.Callback`

Base class for monitoring data histograms in a `LightningModule`. This requires a logger configured in the Trainer, otherwise no data is logged. The specific class that inherits from this base defines what data gets collected.

Parameters `log_every_n_steps` (`Optional[int]`) – The interval at which histograms should be logged. This defaults to the interval defined in the Trainer. Use this to override the Trainer default.

`_is_logger_available` (*logger*)

Return type `bool`

`log_histogram` (*tensor, name*)

Override this method to customize the logging of histograms. Detaches the tensor from the graph and moves it to the CPU for logging.

Parameters

- **`tensor`** (`Tensor`) – The tensor for which to log a histogram
- **`name`** (`str`) – The name of the tensor as determined by the callback. Example: `input/0/[64, 1, 28, 28]`

Return type `None`

`log_histograms` (*batch, group=""*)

Logs the histograms at the interval defined by `row_log_interval`, given a logger is available.

Parameters

- **`batch`** (`Any`) – torch or numpy arrays, or a collection of it (tuple, list, dict, ...), can be nested. If the data appears in a dictionary, the keys are used as labels for the corresponding histogram. Otherwise the histograms get labelled with an integer index. Each label also has the tensors's shape as suffix.
- **`group`** (`str`) – Name under which the histograms will be grouped.

Return type `None`

`on_train_batch_start` (*trainer, pl_module, batch, batch_idx, dataloader_idx*)

Return type `None`

`on_train_start` (*trainer, pl_module*)

Return type `None`

`supported_loggers` = (`pytorch_lightning.loggers.TensorBoardLogger`, `pytorch_lightning.lo`

class `pl_bolts.callbacks.data_monitor.ModuleDataMonitor` (*submodules=None, log_every_n_steps=None*)

Bases: `pl_bolts.callbacks.data_monitor.DataMonitorBase`

Parameters

- **submodules** (`Union[bool, List[str], None]`) – If *True*, logs the in- and output histograms of every submodule in the `LightningModule`, including the root module itself. This parameter can also take a list of names of specific submodules (see example below). Default: *None*, logs only the in- and output of the root module.
- **log_every_n_steps** (`Optional[int]`) – The interval at which histograms should be logged. This defaults to the interval defined in the Trainer. Use this to override the Trainer default.

Note: A too low value for *log_every_n_steps* may have a significant performance impact especially when many submodules are involved, since the logging occurs during the forward pass. It should only be used for debugging purposes.

Example

```
# log the in- and output histograms of the `forward` in LightningModule
trainer = Trainer(callbacks=[ModuleDataMonitor()])

# all submodules in LightningModule
trainer = Trainer(callbacks=[ModuleDataMonitor(submodules=True)])

# specific submodules
trainer = Trainer(callbacks=[ModuleDataMonitor(submodules=["generator",
↪ "generator.conv1"])])
```

_get_submodule_names (*root_module*)

Return type `List[str]`

_register_hook (*module_name*, *module*)

Return type `RemovableHandle`

on_train_end (*trainer*, *pl_module*)

Return type `None`

on_train_start (*trainer*, *pl_module*)

Return type `None`

GROUP_NAME_INPUT = 'input'

GROUP_NAME_OUTPUT = 'output'

class `pl_bolts.callbacks.data_monitor.TrainingDataMonitor` (*log_every_n_steps=None*)

Bases: `pl_bolts.callbacks.data_monitor.DataMonitorBase`

Callback that logs the histogram of values in the batched data passed to *training_step*.

Parameters **log_every_n_steps** (`Optional[int]`) – The interval at which histograms should be logged. This defaults to the interval defined in the Trainer. Use this to override the Trainer default.

Example

```
# log histogram of training data passed to `LightningModule.training_step`  
trainer = Trainer(callbacks=[TrainingDataMonitor()])
```

on_train_batch_start (*trainer*, *pl_module*, *batch*, **args*, ***kwargs*)

Return type None

GROUP_NAME = 'training_step'

`pl_bolts.callbacks.data_monitor.collect_and_name_tensors` (*data*, *output*, *parent_name*=*'input'*)

Recursively fetches all tensors in a (nested) collection of data (depth-first search) and names them. Data in dictionaries get named by their corresponding keys and otherwise they get indexed by an increasing integer. The shape of the tensor gets appended to the name as well.

Parameters

- **data** (*Any*) – A collection of data (potentially nested).
- **output** (*Dict[str, Tensor]*) – A dictionary in which the outputs will be stored.
- **parent_name** (*str*) – Used when called recursively on a nested input data.

Example

```
>>> data = {"x": torch.zeros(2, 3), "y": {"z": torch.zeros(5)}, "w": 1}  
>>> output = {}  
>>> collect_and_name_tensors(data, output)  
>>> output  
{'input/x/[2, 3]': ..., 'input/y/z/[5]': ...}
```

Return type None

`pl_bolts.callbacks.data_monitor.shape2str` (*tensor*)

Returns the shape of a tensor in bracket notation as a string.

Example

```
>>> shape2str(torch.rand(1, 2, 3))  
'[1, 2, 3]'  
>>> shape2str(torch.rand(4))  
'[4]'
```

Return type *str*

pl_bolts.callbacks.knn_online module

class `pl_bolts.callbacks.knn_online.KNNOnlineEvaluator` (*dataset*,
num_classes=None)

Bases: `pytorch_lightning.Callback`

Evaluates self-supervised K nearest neighbors.

Example:

```
# your model must have 1 attribute
model = Model()
model.num_classes = ... # the num of classes in the model

online_eval = KNNOnlineEvaluator(
    num_classes=model.num_classes,
    dataset='imagenet'
)
```

Parameters

- **dataset** (`str`) – if stl10, need to get the labeled batch
- **num_classes** (`Optional[int]`) – Number of classes

get_all_representations (*pl_module*, *dataloader*)

Return type `Tuple[ndarray, ndarray]`

get_representations (*pl_module*, *x*)

Return type `Tensor`

on_validation_epoch_end (*trainer*, *pl_module*)

Return type `None`

to_device (*batch*, *device*)

Return type `Tuple[Tensor, Tensor]`

pl_bolts.callbacks.printing module

class `pl_bolts.callbacks.printing.PrintTableMetricsCallback`

Bases: `pytorch_lightning.callbacks.Callback`

Prints a table with the metrics in columns on every epoch end

Example:

```
from pl_bolts.callbacks import PrintTableMetricsCallback

callback = PrintTableMetricsCallback()
```

Pass into trainer like so:

```
trainer = pl.Trainer(callbacks=[callback])
trainer.fit(...)

# -----
# at the end of every epoch it will print
```

(continues on next page)

(continued from previous page)

```
# -----
# loss|train_loss|val_loss|epoch
# -----
# 2.2541470527648926|2.2541470527648926|2.2158432006835938|0
```

on_epoch_end (trainer, pl_module)

Return type None

pl_bolts.callbacks.printing.**dicts_to_table** (dicts, keys=None, pads=None, fcodes=None, convert_headers=None, header_names=None, skip_none_lines=False, replace_values=None)

Generate ascii table from dictionary Taken from (<https://stackoverflow.com/questions/40056747/print-a-list-of-dictionaries-in-table-form>)

Parameters

- **dicts** (List[Dict]) – input dictionary list; empty lists make keys OR header_names mandatory
- **keys** (Optional[List[str]]) – order list of keys to generate columns for; no key/dict-key should suffix with ‘____’ else adjust code-suffix
- **pads** (Optional[List[str]]) – indicate padding direction and size, eg <10 to right pad alias left-align
- **fcodes** (Optional[List[str]]) – formatting codes for respective column type, eg .3f
- **convert_headers** (Optional[Dict[str, Callable]]) – apply converters(dict) on column keys k, eg timestamps
- **header_names** (Optional[List[str]]) – supply for custom column headers instead of keys
- **skip_none_lines** (bool) – skip line if contains None
- **replace_values** (Optional[Dict[str, Any]]) – specify per column keys k a map from seen value to new value; new value must comply with the columns fcode; CAUTION: modifies input (due speed)

Example

```
>>> a = {'a': 1, 'b': 2}
>>> b = {'a': 3, 'b': 4}
>>> print(dicts_to_table([a, b]))
a|b
--
1|2
3|4
```

Return type str

pl_bolts.callbacks.ssl_online module

```
class pl_bolts.callbacks.ssl_online.SSLOnlineEvaluator(dataset, drop_p=0.2,
                                                    hidden_dim=None,
                                                    z_dim=None,
                                                    num_classes=None)
```

Bases: `pytorch_lightning.Callback`

Attaches a MLP for fine-tuning using the standard self-supervised protocol.

Example:

```
# your model must have 2 attributes
model = Model()
model.z_dim = ... # the representation dim
model.num_classes = ... # the num of classes in the model

online_eval = SSLOnlineEvaluator(
    z_dim=model.z_dim,
    num_classes=model.num_classes,
    dataset='imagenet'
)
```

Parameters

- **dataset** (`str`) – if stl10, need to get the labeled batch
- **drop_p** (`float`) – Dropout probability
- **hidden_dim** (`Optional[int]`) – Hidden dimension for the fine-tune MLP
- **z_dim** (`Optional[int]`) – Representation dimension
- **num_classes** (`Optional[int]`) – Number of classes

get_representations (*pl_module*, *x*)

Return type `Tensor`

on_pretrain_routine_start (*trainer*, *pl_module*)

Return type `None`

on_train_batch_end (*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)

Return type `None`

on_validation_batch_end (*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)

Return type `None`

to_device (*batch*, *device*)

Return type `Tuple[Tensor, Tensor]`

pl_bolts.callbacks.variational module

```
class pl_bolts.callbacks.variational.LatentDimInterpolator(interpolate_epoch_interval=20,  
                                                         range_start=-5,  
                                                         range_end=5,  
                                                         steps=11,  
                                                         num_samples=2,  
                                                         normalize=True)
```

Bases: `pytorch_lightning.callbacks.Callback`

Interpolates the latent space for a model by setting all dims to zero and stepping through the first two dims increasing one unit at a time.

Default interpolates between `[-5, 5]` `(-5, -4, -3, ..., 3, 4, 5)`

Example:

```
from pl_bolts.callbacks import LatentDimInterpolator  
  
Trainer(callbacks=[LatentDimInterpolator()])
```

Parameters

- **interpolate_epoch_interval** (`int`) – default 20
- **range_start** (`int`) – default -5
- **range_end** (`int`) – default 5
- **steps** (`int`) – number of step between start and end
- **num_samples** (`int`) – default 2
- **normalize** (`bool`) – default True (change image to (0, 1) range)

interpolate_latent_space (*pl_module, latent_dim*)

Return type `List[Tensor]`

on_epoch_end (*trainer, pl_module*)

Return type `None`

32.2 pl_bolts.datamodules package

```
class pl_bolts.datamodules.AsynchronousLoader(data, device=torch.device, q_size=10,  
                                              num_batches=None, **kwargs)
```

Bases: `object`

Class for asynchronously loading from CPU memory to device memory with DataLoader.

Note that this only works for single GPU training, multiGPU uses PyTorch’s DataParallel or DistributedDataParallel which uses its own code for transferring data across GPUs. This could just break or make things slower with DataParallel or DistributedDataParallel.

Parameters

- **data** (`Union[DataLoader, Dataset]`) – The PyTorch Dataset or DataLoader we’re using to load.
- **device** (`device`) – The PyTorch device we are loading to

- **q_size** (`int`) – Size of the queue used to store the data loaded to the device
- **num_batches** (`Optional[int]`) – Number of batches to load. This must be set if the dataloader doesn't have a finite `__len__`. It will also override `DataLoader.__len__` if set and `DataLoader` has a `__len__`. Otherwise it can be left as `None`
- ****kwargs** – Any additional arguments to pass to the dataloader if we're constructing one here

`load_instance` (*sample*)

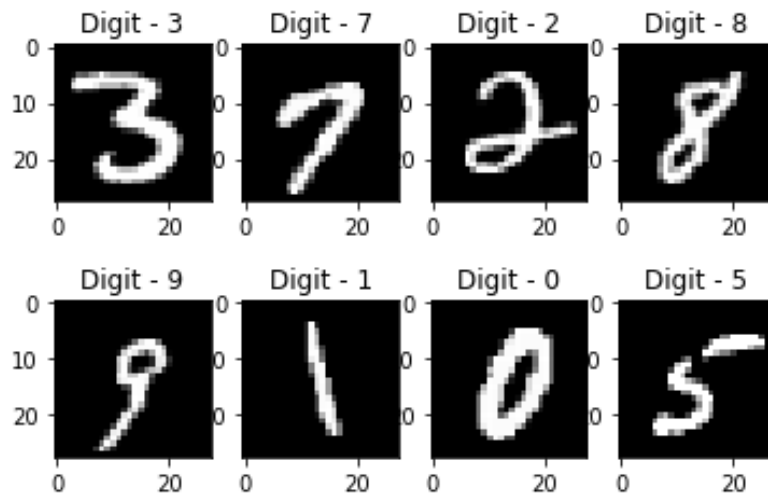
Return type `Any`

`load_loop` ()

Return type `None`

```
class pl_bolts.datamodules.BinaryMNISTDataModule (data_dir=None, val_split=0.2,
                                                  num_workers=16, normalize=False,
                                                  batch_size=32, seed=42, shuffle=False,
                                                  pin_memory=False, drop_last=False, *args, **kwargs)
```

Bases: `pl_bolts.datamodules.vision_datamodule.VisionDataModule`



Specs:

- 10 classes (1 per digit)
- Each image is (1 x 28 x 28)

Binary MNIST, train, val, test splits and transforms

Transforms:

```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor()
])
```

Example:

```

from pl_bolts.datamodules import BinaryMNISTDataModule

dm = BinaryMNISTDataModule('.')
model = LitModel()

Trainer().fit(model, datamodule=dm)

```

Parameters

- **data_dir** (Optional[str]) – Where to save/load the data
- **val_split** (Union[int, float]) – Percent (float) or number (int) of samples to use for the validation split
- **num_workers** (int) – How many workers to use for loading data
- **normalize** (bool) – If true applies image normalize
- **batch_size** (int) – How many samples per batch to load
- **seed** (int) – Random seed to be used for train/val/test splits
- **shuffle** (bool) – If true shuffles the train data every epoch
- **pin_memory** (bool) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (bool) – If true drops the last incomplete batch

dataset_cls

alias of `pl_bolts.datasets.mnist_dataset.BinaryMNIST`

default_transforms()

Default transform for the dataset

Return type `Callable`

dims: tuple = (1, 28, 28)

name: str = 'binary_mnist'

property num_classes

Return: 10

Return type `int`

```

class pl_bolts.datamodules.CIFAR10DataModule (data_dir=None, val_split=0.2,
                                              num_workers=16, normalize=False,
                                              batch_size=32, seed=42, shuffle=False,
                                              pin_memory=False, drop_last=False,
                                              *args, **kwargs)

```

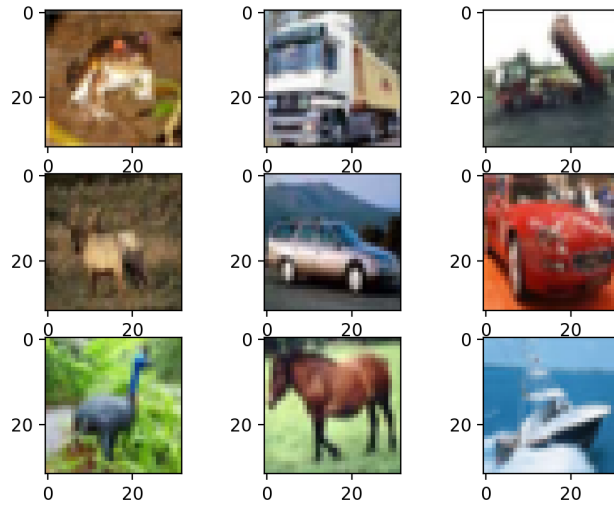
Bases: `pl_bolts.datamodules.vision_datamodule.VisionDataModule`

Specs:

- 10 classes (1 per class)
- Each image is (3 x 32 x 32)

Standard CIFAR10, train, val, test splits and transforms

Transforms:



```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor(),
    transforms.Normalize(
        mean=[x / 255.0 for x in [125.3, 123.0, 113.9]],
        std=[x / 255.0 for x in [63.0, 62.1, 66.7]]
    )
])
```

Example:

```
from pl_bolts.datamodules import CIFAR10DataModule

dm = CIFAR10DataModule(PATH)
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Or you can set your own transforms

Example:

```
dm.train_transforms = ...
dm.test_transforms = ...
dm.val_transforms = ...
```

Parameters

- **data_dir** (Optional[str]) – Where to save/load the data
- **val_split** (Union[int, float]) – Percent (float) or number (int) of samples to use for the validation split
- **num_workers** (int) – How many workers to use for loading data
- **normalize** (bool) – If true applies image normalize
- **batch_size** (int) – How many samples per batch to load

- **seed** (*int*) – Random seed to be used for train/val/test splits
- **shuffle** (*bool*) – If true shuffles the train data every epoch
- **pin_memory** (*bool*) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (*bool*) – If true drops the last incomplete batch

default_transforms ()

Default transform for the dataset

Return type *Callable*

dataset_cls: *type* = *None*

dims: *tuple* = (3, 32, 32)

name: *str* = 'cifar10'

property num_classes

Return: 10

Return type *int*

property num_samples

Return type *int*

```
class pl_bolts.datamodules.TinyCIFAR10DataModule (data_dir=None,          val_split=50,
                                                  num_workers=16,
                                                  num_samples=100, labels=(1, 5,
                                                  8), *args, **kwargs)
```

Bases: *pl_bolts.datamodules.cifar10_datamodule.CIFAR10DataModule*

Standard CIFAR10, train, val, test splits and transforms

Transforms:

```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor(),
    transforms.Normalize(mean=[x / 255.0 for x in [125.3, 123.0, 113.9]],
                          std=[x / 255.0 for x in [63.0, 62.1, 66.7]])
])
```

Example:

```
from pl_bolts.datamodules import CIFAR10DataModule

dm = CIFAR10DataModule(PATH)
model = LitModel(datamodule=dm)
```

Parameters

- **data_dir** (*Optional[str]*) – where to save/load the data
- **val_split** (*int*) – how many of the training images to use for the validation split
- **num_workers** (*int*) – how many workers to use for loading data
- **num_samples** (*int*) – number of examples per selected class/label
- **labels** (*Optional[Sequence]*) – list selected CIFAR10 classes/labels

dataset_cls

alias of `pl_bolts.datasets.cifar10_dataset.TrialCIFAR10`

dims: tuple = (3, 32, 32)

property num_classes

Return number of classes.

Return type int

```
class pl_bolts.datamodules.CityscapesDataModule(data_dir, quality_mode='fine', target_type='instance', num_workers=16, batch_size=32, seed=42, shuffle=False, pin_memory=False, drop_last=False, *args, **kwargs)
```

Bases: `pytorch_lightning.LightningDataModule`



Standard Cityscapes, train, val, test splits and transforms

Note: You need to have downloaded the Cityscapes dataset first and provide the path to where it is saved.

You can download the dataset here: <https://www.cityscapes-dataset.com/>

Specs:

- 30 classes (road, person, sidewalk, etc...)
- (image, target) - image dims: (3 x 1024 x 2048), target dims: (1024 x 2048)

Transforms:

```
transforms = transform_lib.Compose([
    transform_lib.ToTensor(),
    transform_lib.Normalize(
        mean=[0.28689554, 0.32513303, 0.28389177],
        std=[0.18696375, 0.19017339, 0.18720214]
    )
])
```

Example:

```
from pl_bolts.datamodules import CityscapesDataModule

dm = CityscapesDataModule(PATH)
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Or you can set your own transforms

Example:

```
dm.train_transforms = ...
dm.test_transforms = ...
dm.val_transforms = ...
dm.target_transforms = ...
```

Parameters

- **data_dir** (*str*) – where to load the data from path, i.e. where directory leftImg8bit and gtFine or gtCoarse are located
- **quality_mode** (*str*) – the quality mode to use, either ‘fine’ or ‘coarse’
- **target_type** (*str*) – targets to use, either ‘instance’ or ‘semantic’
- **num_workers** (*int*) – how many workers to use for loading data
- **batch_size** (*int*) – number of examples per training/eval step
- **seed** (*int*) – random seed to be used for train/val/test splits
- **shuffle** (*bool*) – If true shuffles the data every epoch
- **pin_memory** (*bool*) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (*bool*) – If true drops the last incomplete batch

_default_target_transforms ()

Return type *Callable*

_default_transforms ()

Return type *Callable*

test_dataloader ()

Cityscapes test set

Return type *DataLoader*

train_dataloader ()

Cityscapes train set

Return type *DataLoader*

val_dataloader ()

Cityscapes val set

Return type *DataLoader*

extra_args: dict = {}

name = 'Cityscapes'

property num_classes

Return: 30

Return type *int*

```
class pl_bolts.datamodules.DiscountedExperienceSource (env, agent, n_steps=1,
                                                    gamma=0.99)
```

Bases: *pl_bolts.datamodules.experience_source.ExperienceSource*

Outputs experiences with a discounted reward over N steps

Args: `env`: Environment that is being used agent: Agent being used to make decisions `n_steps`: Number of steps to return from each environment at once

discount_rewards (*experiences*)

Calculates the discounted reward over N experiences

Parameters `experiences` (`Tuple[Experience]`) – Tuple of Experience

Return type `float`

Returns total discounted reward

runner (*device*)

Iterates through experience tuple and calculate discounted experience

Parameters `device` (`device`) – current device to be used for executing experience steps

Yields Discounted Experience

Return type `Experience`

split_head_tail_exp (*experiences*)

Takes in a tuple of experiences and returns the last state and tail experiences based on if the last state is the end of an episode

Parameters `experiences` (`Tuple[Experience]`) – Tuple of N Experience

Return type `Tuple[List, Tuple[Experience]]`

Returns last state (Array or None) and remaining Experience

class `pl_bolts.datamodules.ExperienceSource` (*env, agent, n_steps=1*)

Bases: `pl_bolts.datamodules.experience_source.BaseExperienceSource`

Experience source class handling single and multiple environment steps

Parameters

- `env` – Environment that is being used
- `agent` – Agent being used to make decisions
- `n_steps` (`int`) – Number of steps to return from each environment at once

env_actions (*device*)

For each environment in the pool, get the correct action :rtype: `List[List[int]]` :returns: List of actions for each env, with size (num_envs, action_size)

env_step (*env_idx, env, action*)

Carries out a step through the given environment using the given action

Parameters

- `env_idx` (`int`) – index of the current environment
- `env` (`object`) – env at index `env_idx`
- `action` (`List[int]`) – action for this environment step

Return type `Experience`

Returns Experience tuple

init_environments ()

For each environment in the pool setups lists for tracking history of size n, state, current reward and current step

Return type None

pop_rewards_steps ()

Returns the list of the current total rewards and steps collected :returns: list of total rewards and steps for all completed episodes for each environment since last pop

pop_total_rewards ()

Returns the list of the current total rewards collected :rtype: `List[float]` :returns: list of total rewards for all completed episodes for each environment since last pop

runner (*device*)

Experience Source iterator yielding Tuple of experiences for n_steps. These come from the pool of environments provided by the user.

Parameters **device** (*device*) – current device to be used for executing experience steps

Return type `Tuple[Experience]`

Returns Tuple of Experiences

update_env_stats (*env_idx*)

To be called at the end of the history tail generation during the termination state. Updates the stats tracked for all environments

Parameters **env_idx** (*int*) – index of the environment used to update stats

Return type None

update_history_queue (*env_idx, exp, history*)

Updates the experience history queue with the latest experiences. In the event of an experience step is in the done state, the history will be incrementally appended to the queue, removing the tail of the history each time.

Parameters

- **env_idx** – index of the environment
- **exp** – the current experience
- **history** – history of experience steps for this environment

Return type None

class `pl_bolts.datamodules.ExperienceSourceDataset` (*generate_batch*)

Bases: `torch.utils.data.IterableDataset`

Basic experience source dataset. Takes a generate_batch function that returns an iterator. The logic for the experience source and how the batch is generated is defined the Lightning model itself

class `pl_bolts.datamodules.FashionMNISTDataModule` (*data_dir=None, val_split=0.2, num_workers=16, normalize=False, batch_size=32, seed=42, shuffle=False, pin_memory=False, drop_last=False, *args, **kwargs*)

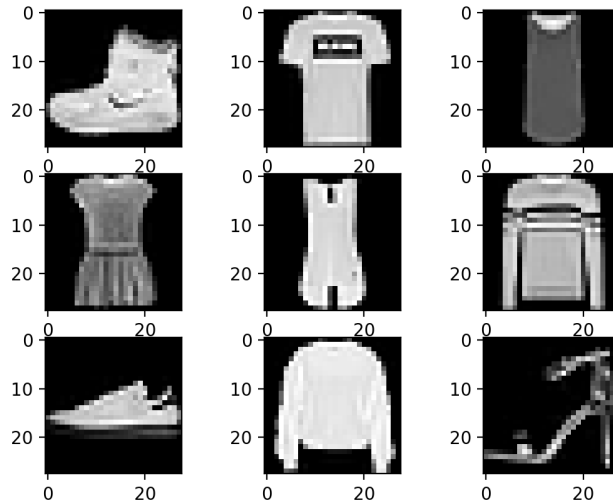
Bases: `pl_bolts.datamodules.vision_datamodule.VisionDataModule`

Specs:

- 10 classes (1 per type)
- Each image is (1 x 28 x 28)

Standard FashionMNIST, train, val, test splits and transforms

Transforms:



```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor()
])
```

Example:

```
from pl_bolts.datamodules import FashionMNISTDataModule

dm = FashionMNISTDataModule('.')
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (Optional[str]) – Where to save/load the data
- **val_split** (Union[int, float]) – Percent (float) or number (int) of samples to use for the validation split
- **num_workers** (int) – How many workers to use for loading data
- **normalize** (bool) – If true applies image normalize
- **batch_size** (int) – How many samples per batch to load
- **seed** (int) – Random seed to be used for train/val/test splits
- **shuffle** (bool) – If true shuffles the train data every epoch
- **pin_memory** (bool) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (bool) – If true drops the last incomplete batch

default_transforms ()

Default transform for the dataset

Return type `Callable`

`dataset_cls:` `type = None`

`dims:` `tuple = (1, 28, 28)`

`name:` `str = 'fashion_mnist'`

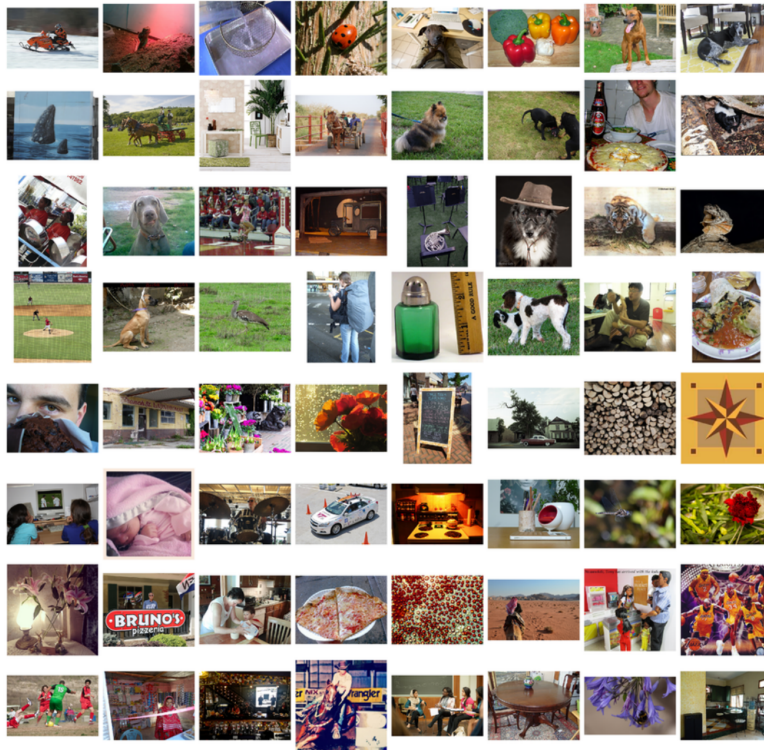
property `num_classes`

Return: 10

Return type `int`

```
class pl_bolts.datamodules.ImagenetDataModule(data_dir, meta_dir=None,
                                             num_imgs_per_val_class=50, image_size=224, num_workers=16,
                                             batch_size=32, shuffle=False,
                                             pin_memory=False, drop_last=False,
                                             *args, **kwargs)
```

Bases: `pytorch_lightning.LightningDataModule`



Specs:

- 1000 classes
- Each image is (3 x varies x varies) (here we default to 3 x 224 x 224)

Imagenet train, val and test dataloaders.

The train set is the imagenet train.

The val set is taken from the train set with `num_imgs_per_val_class` images per class. For example if `num_imgs_per_val_class=2` then there will be 2,000 images in the validation set.

The test set is the official imagenet validation set.

Example:

```
from pl_bolts.datamodules import ImagenetDataModule

dm = ImagenetDataModule(IMGNET_PATH)
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (*str*) – path to the imagenet dataset file
- **meta_dir** (*Optional[str]*) – path to meta.bin file
- **num_imgs_per_val_class** (*int*) – how many images per class for the validation set
- **image_size** (*int*) – final image size
- **num_workers** (*int*) – how many data workers
- **batch_size** (*int*) – batch_size
- **shuffle** (*bool*) – If true shuffles the data every epoch
- **pin_memory** (*bool*) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (*bool*) – If true drops the last incomplete batch

_verify_splits (*data_dir, split*)

Return type *None*

prepare_data ()

This method already assumes you have imagenet2012 downloaded. It validates the data using the meta.bin.

Warning: Please download imagenet on your own first.

Return type *None*

test_dataloader ()

Uses the validation split of imagenet2012 for testing

Return type *DataLoader*

train_dataloader ()

Uses the train split of imagenet2012 and puts away a portion of it for the validation split

Return type *DataLoader*

train_transform ()

The standard imagenet transforms

```
transform_lib.Compose([
    transform_lib.RandomResizedCrop(self.image_size),
    transform_lib.RandomHorizontalFlip(),
    transform_lib.ToTensor(),
```

(continues on next page)

(continued from previous page)

```

        transform_lib.Normalize(
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]
        ),
    ])

```

Return type `Callable`**val_dataloader()**Uses the part of the train split of imagenet2012 that was not used for training via *num_imgs_per_val_class***Parameters**

- **batch_size** – the batch size
- **transforms** – the transforms

Return type `DataLoader`**val_transform()**

The standard imagenet transforms for validation

```

transform_lib.Compose([
    transform_lib.Resize(self.image_size + 32),
    transform_lib.CenterCrop(self.image_size),
    transform_lib.ToTensor(),
    transform_lib.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    ),
])

```

Return type `Callable`**name** = 'imagenet'**property num_classes**

Return:

1000

Return type `int`

```

class pl_bolts.datamodules.KittiDataModule(
    data_dir=None, val_split=0.2, test_split=0.1,
    num_workers=16, batch_size=32, seed=42,
    shuffle=False, pin_memory=False,
    drop_last=False, *args, **kwargs)

```

Bases: `pytorch_lightning.LightningDataModule`

Kitti train, validation and test dataloaders.

Note: You need to have downloaded the Kitti dataset first and provide the path to where it is saved. You can download the dataset here: http://www.cvlibs.net/datasets/kitti/eval_semseg.php?benchmark=semantics2015

Specs:

- 200 samples

- Each image is (3 x 1242 x 376)

In total there are 34 classes but some of these are not useful so by default we use only 19 of the classes specified by the `valid_labels` parameter.

Example:

```
from pl_bolts.datamodules import KittiDataModule

dm = KittiDataModule(PATH)
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (`Optional[str]`) – where to load the data from path, i.e. `‘/path/to/folder/with/data_semantics/’`
- **val_split** (`float`) – size of validation test (default 0.2)
- **test_split** (`float`) – size of test set (default 0.1)
- **num_workers** (`int`) – how many workers to use for loading data
- **batch_size** (`int`) – the batch size
- **seed** (`int`) – random seed to be used for train/val/test splits
- **shuffle** (`bool`) – If true shuffles the data every epoch
- **pin_memory** (`bool`) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (`bool`) – If true drops the last incomplete batch

`_default_transforms()`

Return type `Callable`

`test_dataloader()`

Return type `DataLoader`

`train_dataloader()`

Return type `DataLoader`

`val_dataloader()`

Return type `DataLoader`

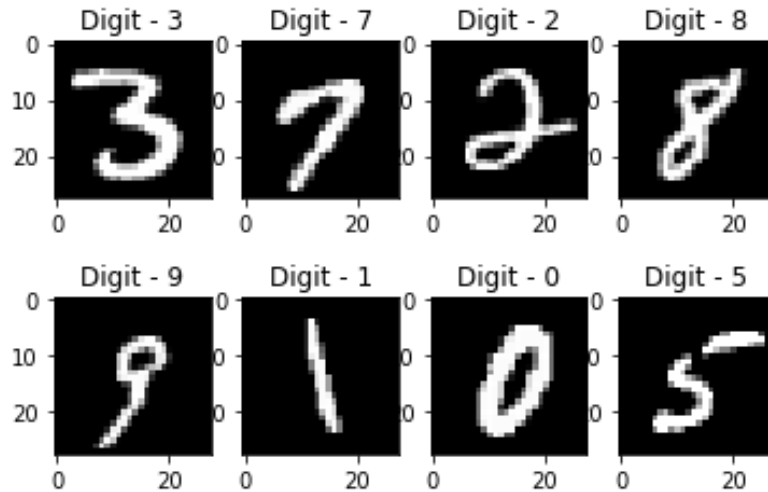
`name = 'kitti'`

```
class pl_bolts.datamodules.MNISTDataModule(data_dir=None, val_split=0.2,
                                           num_workers=16, normalize=False,
                                           batch_size=32, seed=42, shuffle=False,
                                           pin_memory=False, drop_last=False, *args,
                                           **kwargs)
```

Bases: `pl_bolts.datamodules.vision_datamodule.VisionDataModule`

Specs:

- 10 classes (1 per digit)



- Each image is (1 x 28 x 28)

Standard MNIST, train, val, test splits and transforms

Transforms:

```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor()
])
```

Example:

```
from pl_bolts.datamodules import MNISTDataModule

dm = MNISTDataModule('.')
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (`Optional[str]`) – Where to save/load the data
- **val_split** (`Union[int, float]`) – Percent (float) or number (int) of samples to use for the validation split
- **num_workers** (`int`) – How many workers to use for loading data
- **normalize** (`bool`) – If true applies image normalize
- **batch_size** (`int`) – How many samples per batch to load
- **seed** (`int`) – Random seed to be used for train/val/test splits
- **shuffle** (`bool`) – If true shuffles the train data every epoch
- **pin_memory** (`bool`) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (`bool`) – If true drops the last incomplete batch

default_transforms()
Default transform for the dataset

Return type `Callable`

dataset_cls: `type = None`

dims: `tuple = (1, 28, 28)`

name: `str = 'mnist'`

property num_classes

Return: 10

Return type `int`

```
class pl_bolts.datamodules.SklearnDataModule(X, y, x_val=None, y_val=None,
                                             x_test=None, y_test=None, val_split=0.2,
                                             test_split=0.1, num_workers=2,
                                             random_state=1234, shuffle=True,
                                             batch_size=16, pin_memory=False,
                                             drop_last=False, *args, **kwargs)
```

Bases: `pytorch_lightning.LightningDataModule`

Automatically generates the train, validation and test splits for a Numpy dataset. They are set up as dataloaders for convenience. Optionally, you can pass in your own validation and test splits.

Example

```
>>> from sklearn.datasets import load_boston
>>> from pl_bolts.datamodules import SklearnDataModule
...
>>> X, y = load_boston(return_X_y=True)
>>> loaders = SklearnDataModule(X, y, batch_size=32)
...
>>> # train set
>>> train_loader = loaders.train_dataloader()
>>> len(train_loader.dataset)
355
>>> len(train_loader)
12
>>> # validation set
>>> val_loader = loaders.val_dataloader()
>>> len(val_loader.dataset)
100
>>> len(val_loader)
4
>>> # test set
>>> test_loader = loaders.test_dataloader()
>>> len(test_loader.dataset)
51
>>> len(test_loader)
2
```

_init_datasets (*X, y, x_val, y_val, x_test, y_test*)

Return type `None`

test_dataloader()

Return type `DataLoader`

```
train_dataloader()
```

Return type `DataLoader`

```
val_dataloader()
```

Return type `DataLoader`

```
name = 'sklearn'
```

```
class pl_bolts.datamodules.SklearnDataset(X, y, X_transform=None, y_transform=None)
```

Bases: `torch.utils.data.Dataset`

Mapping between numpy (or sklearn) datasets to PyTorch datasets.

Example

```
>>> from sklearn.datasets import load_boston
>>> from pl_bolts.datamodules import SklearnDataset
...
>>> X, y = load_boston(return_X_y=True)
>>> dataset = SklearnDataset(X, y)
>>> len(dataset)
506
```

Parameters

- **X** (`ndarray`) – Numpy ndarray
- **y** (`ndarray`) – Numpy ndarray
- **X_transform** (`Optional[Any]`) – Any transform that works with Numpy arrays
- **y_transform** (`Optional[Any]`) – Any transform that works with Numpy arrays

```
class pl_bolts.datamodules.TensorDataset(X, y, X_transform=None, y_transform=None)
```

Bases: `torch.utils.data.Dataset`

Prepare PyTorch tensor dataset for data loaders.

Example

```
>>> from pl_bolts.datamodules import TensorDataset
...
>>> X = torch.rand(10, 3)
>>> y = torch.rand(10)
>>> dataset = TensorDataset(X, y)
>>> len(dataset)
10
```

Parameters

- **X** (`Tensor`) – PyTorch tensor
- **y** (`Tensor`) – PyTorch tensor
- **X_transform** (`Optional[Any]`) – Any transform that works with PyTorch tensors
- **y_transform** (`Optional[Any]`) – Any transform that works with PyTorch tensors


```
class pl_bolts.datamodules.SSLImagenetDataModule (data_dir, meta_dir=None,
                                                  num_workers=16, batch_size=32,
                                                  shuffle=False, pin_memory=False,
                                                  drop_last=False, *args, **kwargs)
```

Bases: `pytorch_lightning.LightningDataModule`

`_default_transforms()`

Return type `Callable`

`_verify_splits(data_dir, split)`

Return type `None`

`prepare_data()`

Return type `None`

`test_dataloader(num_images_per_class, add_normalize=False)`

Return type `DataLoader`

`train_dataloader(num_images_per_class=-1, add_normalize=False)`

Return type `DataLoader`

`val_dataloader(num_images_per_class=50, add_normalize=False)`

Return type `DataLoader`

`name = 'imagenet'`

property `num_classes`

Return type `int`

```
class pl_bolts.datamodules.STL10DataModule (data_dir=None, unlabeled_val_split=5000,
                                             train_val_split=500, num_workers=16,
                                             batch_size=32, seed=42, shuffle=False,
                                             pin_memory=False, drop_last=False, *args,
                                             **kwargs)
```

Bases: `pytorch_lightning.LightningDataModule`



Specs:

- 10 classes (1 per type)
- Each image is (3 x 96 x 96)

Standard STL-10, train, val, test splits and transforms. STL-10 has support for doing validation splits on the labeled or unlabeled splits

Transforms:

```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor(),
    transforms.Normalize(
        mean=(0.43, 0.42, 0.39),
        std=(0.27, 0.26, 0.27)
    )
])
```

Example:

```
from pl_bolts.datamodules import STL10DataModule

dm = STL10DataModule(PATH)
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (`Optional[str]`) – where to save/load the data
- **unlabeled_val_split** (`int`) – how many images from the unlabeled training split to use for validation
- **train_val_split** (`int`) – how many images from the labeled training split to use for validation
- **num_workers** (`int`) – how many workers to use for loading data
- **batch_size** (`int`) – the batch size
- **seed** (`int`) – random seed to be used for train/val/test splits
- **shuffle** (`bool`) – If true shuffles the data every epoch
- **pin_memory** (`bool`) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (`bool`) – If true drops the last incomplete batch

_default_transforms ()

Return type `Callable`

prepare_data ()

Downloads the unlabeled, train and test split

Return type `None`

test_dataloader ()

Loads the test split of STL10

Parameters

- **batch_size** – the batch size
- **transforms** – the transforms

Return type `DataLoader`

train_dataloader()

Loads the ‘unlabeled’ split minus a portion set aside for validation via *unlabeled_val_split*.

Return type `DataLoader`

train_dataloader_labeled()

Return type `DataLoader`

train_dataloader_mixed()

Loads a portion of the ‘unlabeled’ training data and ‘train’ (labeled) data. both portions have a subset removed for validation via *unlabeled_val_split* and *train_val_split*

Parameters

- **batch_size** – the batch size
- **transforms** – a sequence of transforms

Return type `DataLoader`

val_dataloader()

Loads a portion of the ‘unlabeled’ training data set aside for validation The val dataset = (unlabeled - train_val_split)

Parameters

- **batch_size** – the batch size
- **transforms** – a sequence of transforms

Return type `DataLoader`

val_dataloader_labeled()

Return type `DataLoader`

val_dataloader_mixed()

Loads a portion of the ‘unlabeled’ training data set aside for validation along with the portion of the ‘train’ dataset to be used for validation

unlabeled_val = (unlabeled - train_val_split)

labeled_val = (train- train_val_split)

full_val = unlabeled_val + labeled_val

Parameters

- **batch_size** – the batch size
- **transforms** – a sequence of transforms

Return type `DataLoader`

name = 'stl10'

property num_classes

Return type `int`

```
class pl_bolts.datamodules.VOCDetectionDataModule (data_dir, year='2012',
                                                    num_workers=16,      nor-
                                                    malize=False,        shuf-
                                                    fle=False,      pin_memory=False,
                                                    drop_last=False, *args, **kwargs)
```

Bases: `pytorch_lightning.LightningDataModule`

TODO(teddykoker) docstring

`_default_transforms()`

Return type `Callable`

`prepare_data()`

Saves VOCDetection files to `data_dir`

Return type `None`

`train_dataloader(batch_size=1, image_transforms=None)`

VOCDetection train set uses the *train* subset

Parameters

- **batch_size** (`int`) – size of batch
- **transforms** – custom transforms

Return type `DataLoader`

`val_dataloader(batch_size=1, image_transforms=None)`

VOCDetection val set uses the *val* subset

Parameters

- **batch_size** (`int`) – size of batch
- **transforms** – custom transforms

Return type `DataLoader`

`name = 'vocdetection'`

`property num_classes`

Return: 21

Return type `int`

`class pl_bolts.datamodules.KittiDataset(data_dir, img_size=(1242, 376), void_labels=(0, 1, 2, 3, 4, 5, 6, 9, 10, 14, 15, 16, 18, 29, 30, -1), valid_labels=(7, 8, 11, 12, 13, 17, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 31, 32, 33), transform=None)`

Bases: `torch.utils.data.Dataset`

Note: You need to have downloaded the Kitti dataset first and provide the path to where it is saved. You can download the dataset here: http://www.cvlibs.net/datasets/kitti/eval_semseg.php?benchmark=semantics2015

There are 34 classes, however not all of them are useful for training (e.g. railings on highways). These useless classes (the pixel values of these classes) are stored in *void_labels*. Useful classes are stored in *valid_labels*.

The *encode_segmap* function sets all pixels with any of the *void_labels* to *ignore_index* (250 by default). It also sets all of the valid pixels to the appropriate value between 0 and *len(valid_labels)* (since that is the number of valid classes), so it can be used properly by the loss function when comparing with the output.

Parameters

- **data_dir** (`str`) – where to load the data from path, i.e. `'/path/to/folder/with/data_semantics/'`
- **img_size** (`tuple`) – image dimensions (width, height)
- **void_labels** (`list`) – useless classes to be excluded from training

- **valid_labels** (*list*) – useful classes to include

encode_segmap (*mask*)
Sets void classes to zero so they won't be considered for training

get_filenames (*path*)
Returns a list of absolute paths to images inside given *path*

IMAGE_PATH = 'training/image_2'
MASK_PATH = 'training/semantic'

32.2.1 Submodules

pl_bolts.datamodules.async_data_loader module

```
class pl_bolts.datamodules.async_data_loader.AsynchronousLoader (data, device=torch.device,
                                                                q_size=10,
                                                                num_batches=None,
                                                                **kwargs)
```

Bases: `object`

Class for asynchronously loading from CPU memory to device memory with DataLoader.

Note that this only works for single GPU training, multiGPU uses PyTorch's DataParallel or DistributedDataParallel which uses its own code for transferring data across GPUs. This could just break or make things slower with DataParallel or DistributedDataParallel.

Parameters

- **data** (`Union[DataLoader, Dataset]`) – The PyTorch Dataset or DataLoader we're using to load.
- **device** (`device`) – The PyTorch device we are loading to
- **q_size** (`int`) – Size of the queue used to store the data loaded to the device
- **num_batches** (`Optional[int]`) – Number of batches to load. This must be set if the dataloader doesn't have a finite `__len__`. It will also override `DataLoader.__len__` if set and `DataLoader` has a `__len__`. Otherwise it can be left as `None`
- ****kwargs** – Any additional arguments to pass to the dataloader if we're constructing one here

load_instance (*sample*)

Return type `Any`

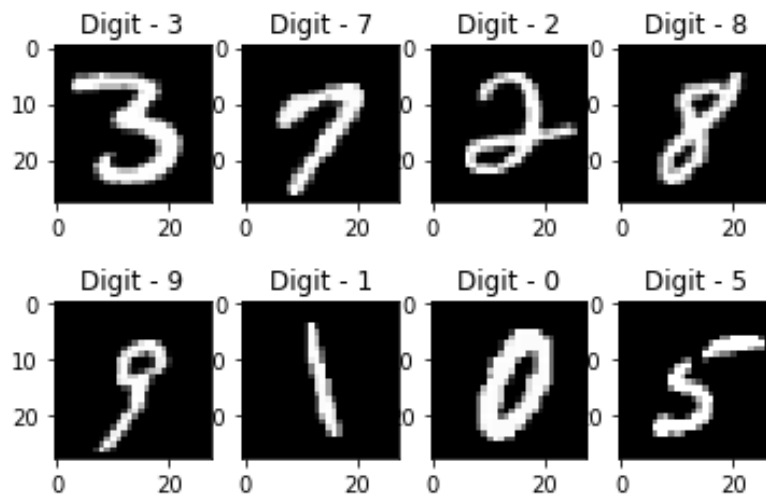
load_loop ()

Return type `None`

pl_bolts.datamodules.binary_mnist_datamodule module

```
class pl_bolts.datamodules.binary_mnist_datamodule.BinaryMNISTDataModule(data_dir=None,
                                                                    val_split=0.2,
                                                                    num_workers=16,
                                                                    normal-
                                                                    ize=False,
                                                                    batch_size=32,
                                                                    seed=42,
                                                                    shuf-
                                                                    fle=False,
                                                                    pin_memory=False,
                                                                    drop_last=False,
                                                                    *args,
                                                                    **kwargs)
```

Bases: `pl_bolts.datamodules.vision_datamodule.VisionDataModule`

**Specs:**

- 10 classes (1 per digit)
- Each image is (1 x 28 x 28)

Binary MNIST, train, val, test splits and transforms

Transforms:

```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor()
])
```

Example:

```
from pl_bolts.datamodules import BinaryMNISTDataModule

dm = BinaryMNISTDataModule('.')
model = LitModel()
```

(continues on next page)

(continued from previous page)

```
Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (`Optional[str]`) – Where to save/load the data
- **val_split** (`Union[int, float]`) – Percent (float) or number (int) of samples to use for the validation split
- **num_workers** (`int`) – How many workers to use for loading data
- **normalize** (`bool`) – If true applies image normalize
- **batch_size** (`int`) – How many samples per batch to load
- **seed** (`int`) – Random seed to be used for train/val/test splits
- **shuffle** (`bool`) – If true shuffles the train data every epoch
- **pin_memory** (`bool`) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (`bool`) – If true drops the last incomplete batch

dataset_cls

alias of `pl_bolts.datasets.mnist_dataset.BinaryMNIST`

default_transforms()

Default transform for the dataset

Return type `Callable`

dims: `tuple = (1, 28, 28)`

name: `str = 'binary_mnist'`

property num_classes

Return: 10

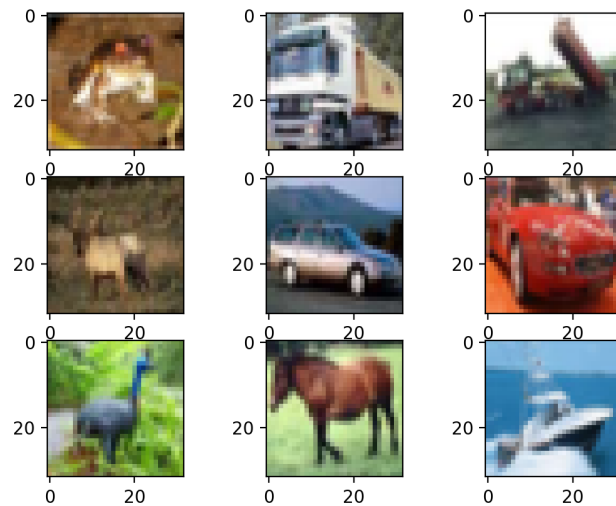
Return type `int`

pl_bolts.datamodules.cifar10_datamodule module

```
class pl_bolts.datamodules.cifar10_datamodule.CIFAR10DataModule(
    data_dir=None,
    val_split=0.2,
    num_workers=16,
    normalize=False,
    batch_size=32,
    seed=42,
    shuffle=False,
    pin_memory=False,
    drop_last=False,
    *args,
    **kwargs)

Bases: pl_bolts.datamodules.vision_datamodule.VisionDataModule
```

Specs:



- 10 classes (1 per class)
- Each image is (3 x 32 x 32)

Standard CIFAR10, train, val, test splits and transforms

Transforms:

```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor(),
    transforms.Normalize(
        mean=[x / 255.0 for x in [125.3, 123.0, 113.9]],
        std=[x / 255.0 for x in [63.0, 62.1, 66.7]]
    )
])
```

Example:

```
from pl_bolts.datamodules import CIFAR10DataModule

dm = CIFAR10DataModule(PATH)
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Or you can set your own transforms

Example:

```
dm.train_transforms = ...
dm.test_transforms = ...
dm.val_transforms = ...
```

Parameters

- **data_dir** (Optional[str]) – Where to save/load the data

- **val_split** (`Union[int, float]`) – Percent (float) or number (int) of samples to use for the validation split
- **num_workers** (`int`) – How many workers to use for loading data
- **normalize** (`bool`) – If true applies image normalize
- **batch_size** (`int`) – How many samples per batch to load
- **seed** (`int`) – Random seed to be used for train/val/test splits
- **shuffle** (`bool`) – If true shuffles the train data every epoch
- **pin_memory** (`bool`) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (`bool`) – If true drops the last incomplete batch

default_transforms ()

Default transform for the dataset

Return type `Callable`

dataset_cls: `type = None`

dims: `tuple = (3, 32, 32)`

name: `str = 'cifar10'`

property num_classes

Return: 10

Return type `int`

property num_samples

Return type `int`

```
class pl_bolts.datamodules.cifar10_datamodule.TinyCIFAR10DataModule (data_dir=None,
                                                                    val_split=50,
                                                                    num_workers=16,
                                                                    num_samples=100,
                                                                    labels=(1,
                                                                    5,      8),
                                                                    *args,
                                                                    **kwargs)
```

Bases: `pl_bolts.datamodules.cifar10_datamodule.CIFAR10DataModule`

Standard CIFAR10, train, val, test splits and transforms

Transforms:

```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor(),
    transforms.Normalize(mean=[x / 255.0 for x in [125.3, 123.0, 113.9]],
                        std=[x / 255.0 for x in [63.0, 62.1, 66.7]])
])
```

Example:

```
from pl_bolts.datamodules import CIFAR10DataModule

dm = CIFAR10DataModule(PATH)
model = LitModel(datamodule=dm)
```

Parameters

- **data_dir** (`Optional[str]`) – where to save/load the data
- **val_split** (`int`) – how many of the training images to use for the validation split
- **num_workers** (`int`) – how many workers to use for loading data
- **num_samples** (`int`) – number of examples per selected class/label
- **labels** (`Optional[Sequence]`) – list selected CIFAR10 classes/labels

dataset_cls

alias of `pl_bolts.datasets.cifar10_dataset.TrialCIFAR10`

dims: `tuple = (3, 32, 32)`

property num_classes

Return number of classes.

Return type `int`

pl_bolts.datamodules.cityscapes_datamodule module

```
class pl_bolts.datamodules.cityscapes_datamodule.CityscapesDataModule(data_dir,
    qual-
    ity_mode='fine',
    tar-
    get_type='instance',
    num_workers=16,
    batch_size=32,
    seed=42,
    shuf-
    fle=False,
    pin_memory=False,
    drop_last=False,
    *args,
    **kwargs)
```

Bases: `pytorch_lightning.LightningDataModule`

Standard Cityscapes, train, val, test splits and transforms

Note: You need to have downloaded the Cityscapes dataset first and provide the path to where it is saved.

You can download the dataset here: <https://www.cityscapes-dataset.com/>

Specs:

- 30 classes (road, person, sidewalk, etc...)
- (image, target) - image dims: (3 x 1024 x 2048), target dims: (1024 x 2048)

Transforms:



```
transforms = transform_lib.Compose([
    transform_lib.ToTensor(),
    transform_lib.Normalize(
        mean=[0.28689554, 0.32513303, 0.28389177],
        std=[0.18696375, 0.19017339, 0.18720214]
    )
])
```

Example:

```
from pl_bolts.datamodules import CityscapesDataModule

dm = CityscapesDataModule(PATH)
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Or you can set your own transforms

Example:

```
dm.train_transforms = ...
dm.test_transforms = ...
dm.val_transforms = ...
dm.target_transforms = ...
```

Parameters

- **data_dir** (`str`) – where to load the data from path, i.e. where directory leftImg8bit and gtFine or gtCoarse are located
- **quality_mode** (`str`) – the quality mode to use, either ‘fine’ or ‘coarse’
- **target_type** (`str`) – targets to use, either ‘instance’ or ‘semantic’
- **num_workers** (`int`) – how many workers to use for loading data
- **batch_size** (`int`) – number of examples per training/eval step
- **seed** (`int`) – random seed to be used for train/val/test splits
- **shuffle** (`bool`) – If true shuffles the data every epoch
- **pin_memory** (`bool`) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them

- **drop_last** (*bool*) – If true drops the last incomplete batch

_default_target_transforms ()

Return type *Callable*

_default_transforms ()

Return type *Callable*

test_dataloader ()

Cityscapes test set

Return type *DataLoader*

train_dataloader ()

Cityscapes train set

Return type *DataLoader*

val_dataloader ()

Cityscapes val set

Return type *DataLoader*

extra_args: *dict* = {}

name = 'Cityscapes'

property num_classes

Return: 30

Return type *int*

pl_bolts.datamodules.experience_source module

Datamodules for RL models that rely on experiences generated during training Based on implementations found here: <https://github.com/Shmuma/ptan/blob/master/ptan/experience.py>

class *pl_bolts.datamodules.experience_source.BaseExperienceSource* (*env, agent*)

Bases: *abc.ABC*

Simplest form of the experience source

Parameters

- **env** – Environment that is being used
- **agent** – Agent being used to make decisions

runner ()

Iterable method that yields steps from the experience source

Return type *Experience*

class *pl_bolts.datamodules.experience_source.DiscountedExperienceSource* (*env, agent, n_steps=1, gamma=0.99*)

Bases: *pl_bolts.datamodules.experience_source.ExperienceSource*

Outputs experiences with a discounted reward over N steps

Args: env: Environment that is being used agent: Agent being used to make decisions n_steps: Number of steps to return from each environment at once

discount_rewards (*experiences*)

Calculates the discounted reward over N experiences

Parameters **experiences** (`Tuple[Experience]`) – Tuple of Experience

Return type `float`

Returns total discounted reward

runner (*device*)

Iterates through experience tuple and calculate discounted experience

Parameters **device** (`device`) – current device to be used for executing experience steps

Yields Discounted Experience

Return type `Experience`

split_head_tail_exp (*experiences*)

Takes in a tuple of experiences and returns the last state and tail experiences based on if the last state is the end of an episode

Parameters **experiences** (`Tuple[Experience]`) – Tuple of N Experience

Return type `Tuple[List, Tuple[Experience]]`

Returns last state (Array or None) and remaining Experience

class `pl_bolts.datamodules.experience_source.Experience` (*state, action, reward, done, new_state*)

Bases: `tuple`

Create new instance of Experience(*state, action, reward, done, new_state*)

__asdict ()

Return a new OrderedDict which maps field names to their values.

classmethod **__make** (*iterable*)

Make a new Experience object from a sequence or iterable

__replace (***kws*)

Return a new Experience object replacing specified fields with new values

__field_defaults = {}

__fields = ('state', 'action', 'reward', 'done', 'new_state')

__fields_defaults = {}

property **action**

Alias for field number 1

property **done**

Alias for field number 3

property **new_state**

Alias for field number 4

property **reward**

Alias for field number 2

property **state**

Alias for field number 0

class `pl_bolts.datamodules.experience_source.ExperienceSource` (*env, agent, n_steps=1*)

Bases: `pl_bolts.datamodules.experience_source.BaseExperienceSource`

Experience source class handling single and multiple environment steps

Parameters

- **env** – Environment that is being used
- **agent** – Agent being used to make decisions
- **n_steps** (*int*) – Number of steps to return from each environment at once

env_actions (*device*)

For each environment in the pool, get the correct action :rtype: `List[List[int]]` :returns: List of actions for each env, with size (num_envs, action_size)

env_step (*env_idx, env, action*)

Carries out a step through the given environment using the given action

Parameters

- **env_idx** (*int*) – index of the current environment
- **env** (*object*) – env at index env_idx
- **action** (`List[int]`) – action for this environment step

Return type *Experience*

Returns Experience tuple

init_environments ()

For each environment in the pool setups lists for tracking history of size n, state, current reward and current step

Return type None

pop_rewards_steps ()

Returns the list of the current total rewards and steps collected :returns: list of total rewards and steps for all completed episodes for each environment since last pop

pop_total_rewards ()

Returns the list of the current total rewards collected :rtype: `List[float]` :returns: list of total rewards for all completed episodes for each environment since last pop

runner (*device*)

Experience Source iterator yielding Tuple of experiences for n_steps. These come from the pool of environments provided by the user.

Parameters **device** (*device*) – current device to be used for executing experience steps

Return type `Tuple[Experience]`

Returns Tuple of Experiences

update_env_stats (*env_idx*)

To be called at the end of the history tail generation during the termination state. Updates the stats tracked for all environments

Parameters **env_idx** (*int*) – index of the environment used to update stats

Return type None

update_history_queue (*env_idx, exp, history*)

Updates the experience history queue with the latest experiences. In the event of an experience step is in the done state, the history will be incrementally appended to the queue, removing the tail of the history each time.

Parameters

- **env_idx** – index of the environment
- **exp** – the current experience
- **history** – history of experience steps for this environment

Return type None

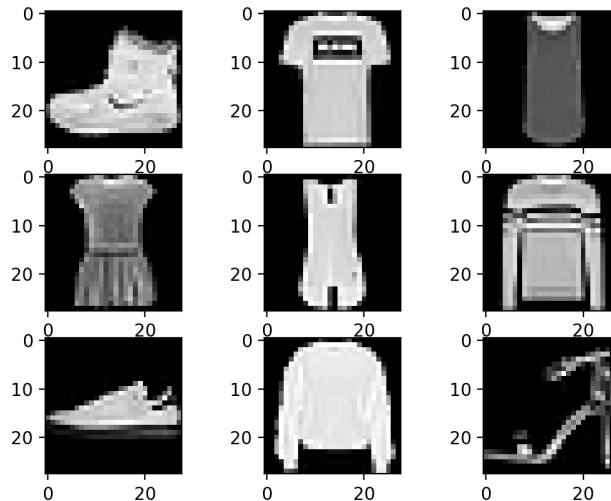
class `pl_bolts.datamodules.experience_source.ExperienceSourceDataset` (*generate_batch*)
 Bases: `torch.utils.data.IterableDataset`

Basic experience source dataset. Takes a `generate_batch` function that returns an iterator. The logic for the experience source and how the batch is generated is defined the Lightning model itself

pl_bolts.datamodules.fashion_mnist_datamodule module

class `pl_bolts.datamodules.fashion_mnist_datamodule.FashionMNISTDataModule` (*data_dir=None, val_split=0.2, num_workers=16, normalize=False, batch_size=32, seed=42, shuffle=False, pin_memory=False, drop_last=False, *args, **kwargs*)

Bases: `pl_bolts.datamodules.vision_datamodule.VisionDataModule`

**Specs:**

- 10 classes (1 per type)

- Each image is (1 x 28 x 28)

Standard FashionMNIST, train, val, test splits and transforms

Transforms:

```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor()
])
```

Example:

```
from pl_bolts.datamodules import FashionMNISTDataModule

dm = FashionMNISTDataModule('.')
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (`Optional[str]`) – Where to save/load the data
- **val_split** (`Union[int, float]`) – Percent (float) or number (int) of samples to use for the validation split
- **num_workers** (`int`) – How many workers to use for loading data
- **normalize** (`bool`) – If true applies image normalize
- **batch_size** (`int`) – How many samples per batch to load
- **seed** (`int`) – Random seed to be used for train/val/test splits
- **shuffle** (`bool`) – If true shuffles the train data every epoch
- **pin_memory** (`bool`) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (`bool`) – If true drops the last incomplete batch

default_transforms ()

Default transform for the dataset

Return type `Callable`

dataset_cls: `type = None`

dims: `tuple = (1, 28, 28)`

name: `str = 'fashion_mnist'`

property num_classes

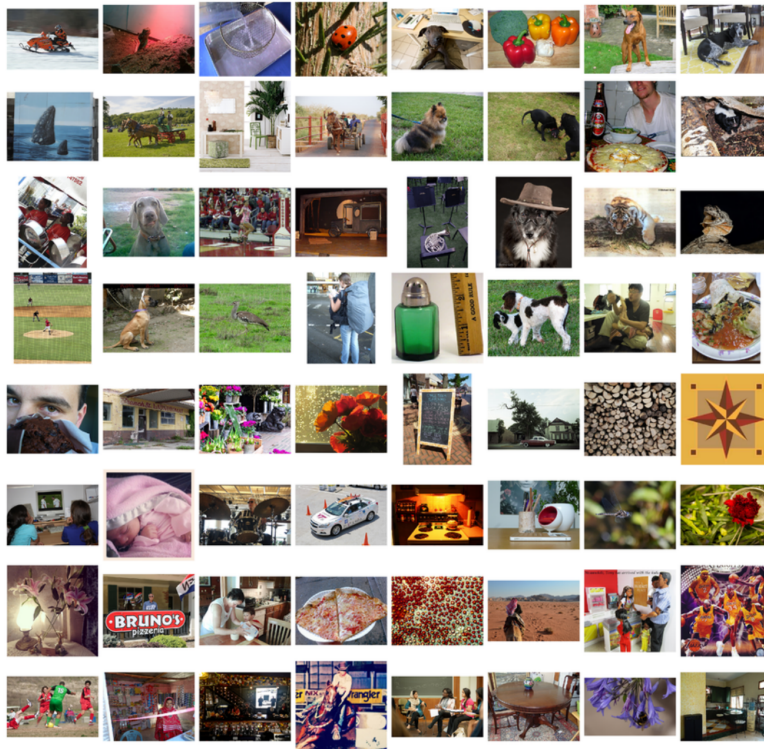
Return: 10

Return type `int`

pl_bolts.datamodules.imagenet_datamodule module

```
class pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule(data_dir,
                                                                meta_dir=None,
                                                                num_imgs_per_val_class=50,
                                                                image_size=224,
                                                                num_workers=16,
                                                                batch_size=32,
                                                                shuffle=False,
                                                                pin_memory=False,
                                                                drop_last=False,
                                                                *args,
                                                                **kwargs)
```

Bases: `pytorch_lightning.LightningDataModule`

**Specs:**

- 1000 classes
- Each image is (3 x varies x varies) (here we default to 3 x 224 x 224)

Imagenet train, val and test dataloaders.

The train set is the imagenet train.

The val set is taken from the train set with `num_imgs_per_val_class` images per class. For example if `num_imgs_per_val_class=2` then there will be 2,000 images in the validation set.

The test set is the official imagenet validation set.

Example:

```
from pl_bolts.datamodules import ImagenetDataModule

dm = ImagenetDataModule(IMGNET_PATH)
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (*str*) – path to the imagenet dataset file
- **meta_dir** (*Optional[str]*) – path to meta.bin file
- **num_imgs_per_val_class** (*int*) – how many images per class for the validation set
- **image_size** (*int*) – final image size
- **num_workers** (*int*) – how many data workers
- **batch_size** (*int*) – batch_size
- **shuffle** (*bool*) – If true shuffles the data every epoch
- **pin_memory** (*bool*) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (*bool*) – If true drops the last incomplete batch

_verify_splits (*data_dir, split*)

Return type *None*

prepare_data ()

This method already assumes you have imagenet2012 downloaded. It validates the data using the meta.bin.

Warning: Please download imagenet on your own first.

Return type *None*

test_dataloader ()

Uses the validation split of imagenet2012 for testing

Return type *DataLoader*

train_dataloader ()

Uses the train split of imagenet2012 and puts away a portion of it for the validation split

Return type *DataLoader*

train_transform ()

The standard imagenet transforms

```
transform_lib.Compose([
    transform_lib.RandomResizedCrop(self.image_size),
    transform_lib.RandomHorizontalFlip(),
    transform_lib.ToTensor(),
    transform_lib.Normalize(
        mean=[0.485, 0.456, 0.406],
```

(continues on next page)

(continued from previous page)

```

        std=[0.229, 0.224, 0.225]
    ),
] )

```

Return type `Callable`**val_dataloader()**Uses the part of the train split of imagenet2012 that was not used for training via *num_imgs_per_val_class***Parameters**

- **batch_size** – the batch size
- **transforms** – the transforms

Return type `DataLoader`**val_transform()**

The standard imagenet transforms for validation

```

transform_lib.Compose([
    transform_lib.Resize(self.image_size + 32),
    transform_lib.CenterCrop(self.image_size),
    transform_lib.ToTensor(),
    transform_lib.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    ),
])

```

Return type `Callable`**name** = 'imagenet'**property num_classes**

Return:

1000

Return type `int`

pl_bolts.datamodules.kitti_datamodule module

```

class pl_bolts.datamodules.kitti_datamodule.KittiDataModule(
    data_dir=None,
    val_split=0.2,
    test_split=0.1,
    num_workers=16,
    batch_size=32,
    seed=42,
    shuffle=True,
    pin_memory=False,
    drop_last=False,
    *args, **kwargs
)

```

Bases: `pytorch_lightning.LightningDataModule`

Kitti train, validation and test dataloaders.

Note: You need to have downloaded the Kitti dataset first and provide the path to where it is saved. You can download the dataset here: http://www.cvlibs.net/datasets/kitti/eval_semseg.php?benchmark=semantics2015

Specs:

- 200 samples
- Each image is (3 x 1242 x 376)

In total there are 34 classes but some of these are not useful so by default we use only 19 of the classes specified by the `valid_labels` parameter.

Example:

```
from pl_bolts.datamodules import KittiDataModule

dm = KittiDataModule(PATH)
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (`Optional[str]`) – where to load the data from path, i.e. `'/path/to/folder/with/data_semantics/'`
- **val_split** (`float`) – size of validation test (default 0.2)
- **test_split** (`float`) – size of test set (default 0.1)
- **num_workers** (`int`) – how many workers to use for loading data
- **batch_size** (`int`) – the batch size
- **seed** (`int`) – random seed to be used for train/val/test splits
- **shuffle** (`bool`) – If true shuffles the data every epoch
- **pin_memory** (`bool`) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (`bool`) – If true drops the last incomplete batch

`_default_transforms()`

Return type `Callable`

`test_dataloader()`

Return type `DataLoader`

`train_dataloader()`

Return type `DataLoader`

`val_dataloader()`

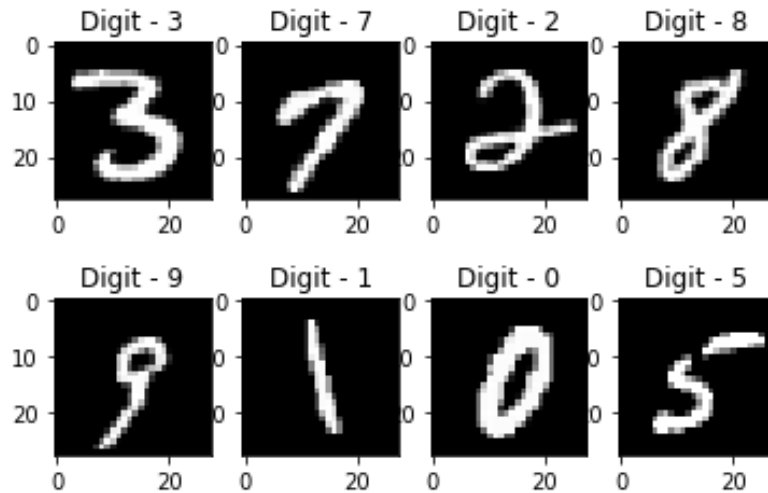
Return type `DataLoader`

`name = 'kitti'`

pl_bolts.datamodules.mnist_datamodule module

```
class pl_bolts.datamodules.mnist_datamodule.MNISTDataModule (data_dir=None,
                                                             val_split=0.2,
                                                             num_workers=16,
                                                             normalize=False,
                                                             batch_size=32,
                                                             seed=42,      shuf-
                                                             fle=False,
                                                             pin_memory=False,
                                                             drop_last=False,
                                                             *args, **kwargs)
```

Bases: `pl_bolts.datamodules.vision_datamodule.VisionDataModule`

**Specs:**

- 10 classes (1 per digit)
- Each image is (1 x 28 x 28)

Standard MNIST, train, val, test splits and transforms

Transforms:

```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor()
])
```

Example:

```
from pl_bolts.datamodules import MNISTDataModule

dm = MNISTDataModule('.')
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (`Optional[str]`) – Where to save/load the data
- **val_split** (`Union[int, float]`) – Percent (float) or number (int) of samples to use for the validation split
- **num_workers** (`int`) – How many workers to use for loading data
- **normalize** (`bool`) – If true applies image normalize
- **batch_size** (`int`) – How many samples per batch to load
- **seed** (`int`) – Random seed to be used for train/val/test splits
- **shuffle** (`bool`) – If true shuffles the train data every epoch
- **pin_memory** (`bool`) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (`bool`) – If true drops the last incomplete batch

default_transforms ()

Default transform for the dataset

Return type `Callable`

dataset_cls: `type = None`

dims: `tuple = (1, 28, 28)`

name: `str = 'mnist'`

property num_classes

Return: 10

Return type `int`

pl_bolts.datamodules.sklearn_datamodule module

```
class pl_bolts.datamodules.sklearn_datamodule.SklearnDataModule (X, y,
    x_val=None,
    y_val=None,
    x_test=None,
    y_test=None,
    val_split=0.2,
    test_split=0.1,
    num_workers=2,
    random_state=1234,
    shuffle=True,
    batch_size=16,
    pin_memory=False,
    drop_last=False,
    *args,
    **kwargs)
```

Bases: `pytorch_lightning.LightningDataModule`

Automatically generates the train, validation and test splits for a Numpy dataset. They are set up as dataloaders for convenience. Optionally, you can pass in your own validation and test splits.

Example

```

>>> from sklearn.datasets import load_boston
>>> from pl_bolts.datamodules import SklearnDataModule
...
>>> X, y = load_boston(return_X_y=True)
>>> loaders = SklearnDataModule(X, y, batch_size=32)
...
>>> # train set
>>> train_loader = loaders.train_dataloader()
>>> len(train_loader.dataset)
355
>>> len(train_loader)
12
>>> # validation set
>>> val_loader = loaders.val_dataloader()
>>> len(val_loader.dataset)
100
>>> len(val_loader)
4
>>> # test set
>>> test_loader = loaders.test_dataloader()
>>> len(test_loader.dataset)
51
>>> len(test_loader)
2

```

`_init_datasets` (*X*, *y*, *x_val*, *y_val*, *x_test*, *y_test*)

Return type `None`

`test_dataloader` ()

Return type `DataLoader`

`train_dataloader` ()

Return type `DataLoader`

`val_dataloader` ()

Return type `DataLoader`

`name` = `'sklearn'`

```

class pl_bolts.datamodules.sklearn_datamodule.SklearnDataset (X, y,
                                                                X_transform=None,
                                                                y_transform=None)

```

Bases: `torch.utils.data.Dataset`

Mapping between numpy (or sklearn) datasets to PyTorch datasets.

Example

```
>>> from sklearn.datasets import load_boston
>>> from pl_bolts.datamodules import SklearnDataset
...
>>> X, y = load_boston(return_X_y=True)
>>> dataset = SklearnDataset(X, y)
>>> len(dataset)
506
```

Parameters

- **X** (`ndarray`) – Numpy ndarray
- **y** (`ndarray`) – Numpy ndarray
- **X_transform** (`Optional[Any]`) – Any transform that works with Numpy arrays
- **y_transform** (`Optional[Any]`) – Any transform that works with Numpy arrays

```
class pl_bolts.datamodules.sklearn_datamodule.TensorDataset(X, y,
                                                             X_transform=None,
                                                             y_transform=None)
```

Bases: `torch.utils.data.Dataset`

Prepare PyTorch tensor dataset for data loaders.

Example

```
>>> from pl_bolts.datamodules import TensorDataset
...
>>> X = torch.rand(10, 3)
>>> y = torch.rand(10)
>>> dataset = TensorDataset(X, y)
>>> len(dataset)
10
```

Parameters

- **X** (`Tensor`) – PyTorch tensor
- **y** (`Tensor`) – PyTorch tensor
- **X_transform** (`Optional[Any]`) – Any transform that works with PyTorch tensors
- **y_transform** (`Optional[Any]`) – Any transform that works with PyTorch tensors

pl_bolts.datamodules.ssl_imagenet_datamodule module

```

class pl_bolts.datamodules.ssl_imagenet_datamodule.SSLImagenetDataModule (data_dir,
                                                                    meta_dir=None,
                                                                    num_workers=16,
                                                                    batch_size=32,
                                                                    shuffle=True,
                                                                    pin_memory=False,
                                                                    drop_last=False,
                                                                    *args,
                                                                    **kwargs)

Bases: pytorch_lightning.LightningDataModule

default_transforms ()
    Return type Callable

verify_splits (data_dir, split)
    Return type None

prepare_data ()
    Return type None

test_dataloader (num_images_per_class, add_normalize=False)
    Return type DataLoader

train_dataloader (num_images_per_class=-1, add_normalize=False)
    Return type DataLoader

val_dataloader (num_images_per_class=50, add_normalize=False)
    Return type DataLoader

name = 'imagenet'

property num_classes
    Return type int

```

pl_bolts.datamodules.stl10_datamodule module

```

class pl_bolts.datamodules.stl10_datamodule.STL10DataModule (data_dir=None,
                                                                unlabeled_val_split=5000,
                                                                train_val_split=500,
                                                                num_workers=16,
                                                                batch_size=32,
                                                                seed=42,
                                                                shuffle=True,
                                                                pin_memory=False,
                                                                drop_last=False,
                                                                *args, **kwargs)

Bases: pytorch_lightning.LightningDataModule

Specs:
    • 10 classes (1 per type)

```



- Each image is (3 x 96 x 96)

Standard STL-10, train, val, test splits and transforms. STL-10 has support for doing validation splits on the labeled or unlabeled splits

Transforms:

```
mnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor(),
    transforms.Normalize(
        mean=(0.43, 0.42, 0.39),
        std=(0.27, 0.26, 0.27)
    )
])
```

Example:

```
from pl_bolts.datamodules import STL10DataModule

dm = STL10DataModule(PATH)
model = LitModel()

Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (Optional[str]) – where to save/load the data
- **unlabeled_val_split** (int) – how many images from the unlabeled training split to use for validation
- **train_val_split** (int) – how many images from the labeled training split to use for validation
- **num_workers** (int) – how many workers to use for loading data
- **batch_size** (int) – the batch size
- **seed** (int) – random seed to be used for train/val/test splits
- **shuffle** (bool) – If true shuffles the data every epoch
- **pin_memory** (bool) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (bool) – If true drops the last incomplete batch

`_default_transforms()`

Return type `Callable`

`prepare_data()`

Downloads the unlabeled, train and test split

Return type `None`

`test_dataloader()`

Loads the test split of STL10

Parameters

- **batch_size** – the batch size
- **transforms** – the transforms

Return type `DataLoader`

`train_dataloader()`

Loads the ‘unlabeled’ split minus a portion set aside for validation via *unlabeled_val_split*.

Return type `DataLoader`

`train_dataloader_labeled()`

Return type `DataLoader`

`train_dataloader_mixed()`

Loads a portion of the ‘unlabeled’ training data and ‘train’ (labeled) data. both portions have a subset removed for validation via *unlabeled_val_split* and *train_val_split*

Parameters

- **batch_size** – the batch size
- **transforms** – a sequence of transforms

Return type `DataLoader`

`val_dataloader()`

Loads a portion of the ‘unlabeled’ training data set aside for validation The val dataset = (unlabeled - train_val_split)

Parameters

- **batch_size** – the batch size
- **transforms** – a sequence of transforms

Return type `DataLoader`

`val_dataloader_labeled()`

Return type `DataLoader`

`val_dataloader_mixed()`

Loads a portion of the ‘unlabeled’ training data set aside for validation along with the portion of the ‘train’ dataset to be used for validation

`unlabeled_val = (unlabeled - train_val_split)`

`labeled_val = (train - train_val_split)`

`full_val = unlabeled_val + labeled_val`

Parameters

- **batch_size** – the batch size
- **transforms** – a sequence of transforms

Return type `DataLoader`

name = 'stl10'

property num_classes

Return type `int`

`pl_bolts.datamodules.vision_datamodule` module

```
class pl_bolts.datamodules.vision_datamodule.VisionDataModule(data_dir=None,  
                                                             val_split=0.2,  
                                                             num_workers=16,  
                                                             normalize=False,  
                                                             batch_size=32,  
                                                             seed=42, shuffle=False,  
                                                             pin_memory=False,  
                                                             drop_last=False,  
                                                             *args, **kwargs)
```

Bases: `pytorch_lightning.LightningDataModule`

Parameters

- **data_dir** (`Optional[str]`) – Where to save/load the data
- **val_split** (`Union[int, float]`) – Percent (float) or number (int) of samples to use for the validation split
- **num_workers** (`int`) – How many workers to use for loading data
- **normalize** (`bool`) – If true applies image normalize
- **batch_size** (`int`) – How many samples per batch to load
- **seed** (`int`) – Random seed to be used for train/val/test splits
- **shuffle** (`bool`) – If true shuffles the train data every epoch
- **pin_memory** (`bool`) – If true, the data loader will copy Tensors into CUDA pinned memory before returning them
- **drop_last** (`bool`) – If true drops the last incomplete batch

_data_loader (*dataset, shuffle=False*)

Return type `DataLoader`

_get_splits (*len_dataset*)

Computes split lengths for train and validation set

Return type `List[int]`

_split_dataset (*dataset, train=True*)

Splits the dataset into train and validation set

Return type `Dataset`

abstract default_transforms ()

Default transform for the dataset

Return type `Callable`

prepare_data (**args, **kwargs*)
Saves files to data_dir

Return type `None`

setup (*stage=None*)
Creates train, val, and test dataset

Return type `None`

test_dataloader (**args, **kwargs*)
The test dataloader

Return type `Union[DataLoader, List[DataLoader]]`

train_dataloader (**args, **kwargs*)
The train dataloader

Return type `DataLoader`

val_dataloader (**args, **kwargs*)
The val dataloader

Return type `Union[DataLoader, List[DataLoader]]`

EXTRA_ARGS: `dict = {}`

dataset_cls: `type = None`
Dataset class to use

dims: `tuple = None`
A tuple describing the shape of the data

name: `str = ''`

pl_bolts.datamodules.vocdetection_datamodule module

class `pl_bolts.datamodules.vocdetection_datamodule.Compose` (*transforms, image_transforms=None*)
Bases: `object`
Like `torchvision.transforms.compose` but works for (image, target)

__call__ (*image, target*)
Call self as a function.

Return type `Tuple[Tensor, Tensor]`

class `pl_bolts.datamodules.vocdetection_datamodule.VOCDetectionDataModule` (*data_dir, year='2012', num_workers=16, normalize=False, shuffle=False, pin_memory=False, drop_last=False, *args, **kwargs*)
Bases: `pytorch_lightning.LightningDataModule`

TODO(teddykoker) docstring

`_default_transforms()`

Return type `Callable`

`prepare_data()`

Saves VOCDetection files to `data_dir`

Return type `None`

`train_dataloader(batch_size=1, image_transforms=None)`

VOCDetection train set uses the *train* subset

Parameters

- `batch_size(int)` – size of batch
- `transforms` – custom transforms

Return type `DataLoader`

`val_dataloader(batch_size=1, image_transforms=None)`

VOCDetection val set uses the *val* subset

Parameters

- `batch_size(int)` – size of batch
- `transforms` – custom transforms

Return type `DataLoader`

`name = 'vocdetection'`

`property num_classes`

Return: 21

Return type `int`

`pl_bolts.datamodules.vocdetection_datamodule._collate_fn(batch)`

Return type `tuple`

`pl_bolts.datamodules.vocdetection_datamodule._prepare_voc_instance(image, target)`

Prepares VOC dataset into appropriate target for fasterrcnn

<https://github.com/pytorch/vision/issues/1097#issuecomment-508917489>

32.3 pl_bolts.datasets package

`class pl_bolts.datasets.LightDataset(*args, **kwargs)`

Bases: `abc.ABC`, `torch.utils.data.Dataset`

`_download_from_url(base_url, data_folder, file_name)`

`static _prepare_subset(full_data, full_targets, num_samples, labels)`

Prepare a subset of a common dataset.

Return type `Tuple[Tensor, Tensor]`

`DATASET_NAME = 'light'`

`cache_folder_name: str = None`

property `cached_folder_path`

Return type `str`

`data: torch.Tensor = None`

`dir_path: str = None`

`normalize: tuple = None`

`targets: torch.Tensor = None`

`class pl_bolts.datasets.CIFAR10 (data_dir='.', train=True, transform=None, download=True)`
 Bases: `pl_bolts.datasets.base_dataset.LightDataset`

Customized `CIFAR10` dataset for testing Pytorch Lightning without the torchvision dependency.

Part of the code was copied from <https://github.com/pytorch/vision/blob/build/v0.5.0/torchvision/datasets/>

Parameters

- **data_dir** (`str`) – Root directory of dataset where `CIFAR10/processed/training.pt` and `CIFAR10/processed/test.pt` exist.
- **train** (`bool`) – If True, creates dataset from `training.pt`, otherwise from `test.pt`.
- **download** (`bool`) – If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.

Examples

```
>>> from torchvision import transforms
>>> from pl_bolts.transforms.dataset_normalizations import cifar10_normalization
>>> cf10_transforms = transforms.Compose([transforms.ToTensor(), cifar10_
↳ normalization()])
>>> dataset = CIFAR10(download=True, transform=cf10_transforms, data_dir="datasets
↳ ")
>>> len(dataset)
50000
>>> torch.bincount(dataset.targets)
tensor([5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000])
>>> data, label = dataset[0]
>>> data.shape
torch.Size([3, 32, 32])
>>> label
6
```

Labels:

```
airplane: 0
automobile: 1
bird: 2
cat: 3
deer: 4
dog: 5
frog: 6
horse: 7
ship: 8
truck: 9
```

```
classmethod _check_exists (data_folder, file_names)
```

Return type `bool`

```
_extract_archive_save_torch (download_path)
```

```
_unpickle (path_folder, file_name)
```

Return type `Tuple[Tensor, Tensor]`

```
download (data_folder)
```

Download the data if it doesn't exist in `cached_folder_path` already.

Return type `None`

```
prepare_data (download)
```

```
BASE_URL = 'https://www.cs.toronto.edu/~kriz/'
```

```
DATASET_NAME = 'CIFAR10'
```

```
FILE_NAME = 'cifar-10-python.tar.gz'
```

```
TEST_FILE_NAME = 'test.pt'
```

```
TRAIN_FILE_NAME = 'training.pt'
```

```
cache_folder_name: str = 'complete'
```

```
data = None
```

```
dir_path = None
```

```
labels = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
normalize = None
```

```
relabel = False
```

```
targets = None
```

```
class pl_bolts.datasets.TrialCIFAR10 (data_dir='.', train=True, transform=None, download=False, num_samples=100, labels=(1, 5, 8), relabel=True)
```

Bases: `pl_bolts.datasets.cifar10_dataset.CIFAR10`

Customized [CIFAR10](#) dataset for testing Pytorch Lightning without the torchvision dependency. .. rubric:: Examples

```
>>> dataset = TrialCIFAR10(download=True, num_samples=150, labels=(1, 5, 8), data_dir="datasets")
>>> len(dataset)
450
>>> sorted(set([d.item() for d in dataset.targets]))
[1, 5, 8]
>>> torch.bincount(dataset.targets)
tensor([ 0, 150,  0,  0,  0, 150,  0,  0, 150])
>>> data, label = dataset[0]
>>> data.shape
torch.Size([3, 32, 32])
```

Parameters

- **data_dir** (`str`) – Root directory of dataset where `CIFAR10/processed/training.pt` and `CIFAR10/processed/test.pt` exist.

- **train** (*bool*) – If True, creates dataset from `training.pt`, otherwise from `test.pt`.
- **download** (*bool*) – If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.
- **num_samples** (*int*) – number of examples per selected class/digit
- **labels** (*Optional[Sequence]*) – list selected CIFAR10 digits/classes

prepare_data (*download*)

Return type `None`

data = `None`

dir_path = `None`

normalize = `None`

targets = `None`

class `pl_bolts.datasets.ConcatDataset` (**datasets*)

Bases: `torch.utils.data.Dataset`

class `pl_bolts.datasets.DummyDataset` (**shapes, num_samples=10000*)

Bases: `torch.utils.data.Dataset`

Generate a dummy dataset

Example

```
>>> from pl_bolts.datasets import DummyDataset
>>> from torch.utils.data import DataLoader
>>> # mnist dims
>>> ds = DummyDataset((1, 28, 28), (1, ))
>>> dl = DataLoader(ds, batch_size=7)
>>> # get first batch
>>> batch = next(iter(dl))
>>> x, y = batch
>>> x.size()
torch.Size([7, 1, 28, 28])
>>> y.size()
torch.Size([7, 1])
```

Parameters

- ***shapes** – list of shapes
- **num_samples** (*int*) – how many samples to use in this dataset

class `pl_bolts.datasets.DummyDetectionDataset` (*img_shape=(3, 256, 256), num_boxes=1,*

num_classes=2, num_samples=10000)

Bases: `torch.utils.data.Dataset`

Generate a dummy dataset for detection

Example

```
>>> from pl_bolts.datasets import DummyDetectionDataset
>>> from torch.utils.data import DataLoader
>>> ds = DummyDetectionDataset()
>>> dl = DataLoader(ds, batch_size=7)
```

Parameters

- ***shapes** – list of shapes
- **num_samples** (*int*) – how many samples to use in this dataset

`_random_bbox()`

class `pl_bolts.datasets.RandomDataset` (*size*, *num_samples=250*)
Bases: `torch.utils.data.Dataset`

Generate a dummy dataset

Example

```
>>> from pl_bolts.datasets import RandomDataset
>>> from torch.utils.data import DataLoader
>>> ds = RandomDataset(10)
>>> dl = DataLoader(ds, batch_size=7)
```

Parameters

- **size** (*int*) – tuple
- **num_samples** (*int*) – number of samples

class `pl_bolts.datasets.RandomDictDataset` (*size*, *num_samples=250*)
Bases: `torch.utils.data.Dataset`

Generate a dummy dataset with a dict structure

Example

```
>>> from pl_bolts.datasets import RandomDictDataset
>>> from torch.utils.data import DataLoader
>>> ds = RandomDictDataset(10)
>>> dl = DataLoader(ds, batch_size=7)
```

Parameters

- **size** (*int*) – tuple
- **num_samples** (*int*) – number of samples

class `pl_bolts.datasets.RandomDictStringDataset` (*size*, *num_samples=250*)
Bases: `torch.utils.data.Dataset`

Generate a dummy dataset with strings

Example

```
>>> from pl_bolts.datasets import RandomDictStringDataset
>>> from torch.utils.data import DataLoader
>>> ds = RandomDictStringDataset(10)
>>> dl = DataLoader(ds, batch_size=7)
```

Parameters

- **size** (*int*) – tuple
- **num_samples** (*int*) – number of samples

```
class pl_bolts.datasets.UnlabeledImagenet (root,          split='train',      num_classes=-1,
                                           num_imgs_per_class=-1,
                                           num_imgs_per_class_val_split=50,
                                           meta_dir=None, **kwargs)
```

Bases: `object`

Official train set gets split into train, val. (using `nb_imgs_per_val_class` for each class). Official validation becomes test set

Within each class, we further allow limiting the number of samples per class (for semi-sup lng)

Parameters

- **root** – path of dataset
- **split** (*str*) –
- **num_classes** (*int*) – Sets the limit of classes
- **num_imgs_per_class** (*int*) – Limits the number of images per class
- **num_imgs_per_class_val_split** (*int*) – How many images per class to generate the val split
- **download** –
- **kwargs** –

```
classmethod generate_meta_bins (devkit_dir)
```

```
partition_train_set (imgs, nb_imgs_in_val)
```

```
class pl_bolts.datasets.KittiDataset (data_dir, img_size=(1242, 376), void_labels=(0, 1,
                                           2, 3, 4, 5, 6, 9, 10, 14, 15, 16, 18, 29, 30, -1),
                                       valid_labels=(7, 8, 11, 12, 13, 17, 19, 20, 21, 22, 23,
                                                    24, 25, 26, 27, 28, 31, 32, 33), transform=None)
```

Bases: `torch.utils.data.Dataset`

Note: You need to have downloaded the Kitti dataset first and provide the path to where it is saved. You can download the dataset here: http://www.cvlibs.net/datasets/kitti/eval_semseg.php?benchmark=semantics2015

There are 34 classes, however not all of them are useful for training (e.g. railings on highways). These useless classes (the pixel values of these classes) are stored in `void_labels`. Useful classes are stored in `valid_labels`.

The `encode_segmap` function sets all pixels with any of the `void_labels` to `ignore_index` (250 by default). It also sets all of the valid pixels to the appropriate value between 0 and `len(valid_labels)` (since that is the number of valid classes), so it can be used properly by the loss function when comparing with the output.

Parameters

- **data_dir** (*str*) – where to load the data from path, i.e. `'/path/to/folder/with/data_semantics/'`
- **img_size** (*tuple*) – image dimensions (width, height)
- **void_labels** (*list*) – useless classes to be excluded from training
- **valid_labels** (*list*) – useful classes to include

encode_segmap (*mask*)

Sets void classes to zero so they won't be considered for training

get_filenames (*path*)

Returns a list of absolute paths to images inside given *path*

IMAGE_PATH = `'training/image_2'`

MASK_PATH = `'training/semantic'`

class `pl_bolts.datasets.BinaryMNIST`

Bases: `object`

class `pl_bolts.datasets.CIFAR10Mixed` (*root*, *split*='val', *transform*=None, *target_transform*=None, *download*=False, *nb_labeled_per_class*=None, *val_pct*=0.1)

Bases: `pl_bolts.datasets.ssl_anddim_datasets.SSLDatasetMixin`, `object`

class `pl_bolts.datasets.SSLDatasetMixin`

Bases: `abc.ABC`

classmethod `deterministic_shuffle` (*x*, *y*)

classmethod `generate_train_val_split` (*examples*, *labels*, *pct_val*)

Splits dataset uniformly across classes

classmethod `select_nb_imgs_per_class` (*examples*, *labels*, *nb_imgs_in_val*)

Splits a dataset into two parts. The labeled split has *nb_imgs_in_val* per class

`pl_bolts.datasets.extract_archive` (*from_path*, *to_path*=None, *remove_finished*=False)

`pl_bolts.datasets.parse_devkit_archive` (*root*, *file*=None)

Parse the devkit archive of the ImageNet2012 classification dataset and save the meta information in a binary file.

Parameters

- **root** (*str*) – Root directory containing the devkit archive
- **file** (*str*, *optional*) – Name of devkit archive. Defaults to `'ILSVRC2012_devkit_t12.tar.gz'`

32.3.1 Submodules

pl_bolts.datasets.base_dataset module

```
class pl_bolts.datasets.base_dataset.LightDataset(*args, **kwargs)
    Bases: abc.ABC, torch.utils.data.Dataset

    _download_from_url(base_url, data_folder, file_name)

    static _prepare_subset(full_data, full_targets, num_samples, labels)
        Prepare a subset of a common dataset.

        Return type Tuple[torch.Tensor, torch.Tensor]

    DATASET_NAME = 'light'

    cache_folder_name: str = None

    property cached_folder_path
        Return type str

    data: torch.Tensor = None

    dir_path: str = None

    normalize: tuple = None

    targets: torch.Tensor = None
```

pl_bolts.datasets.cifar10_dataset module

```
class pl_bolts.datasets.cifar10_dataset.CIFAR10(data_dir='.', train=True, trans-
                                                form=None, download=True)
    Bases: pl_bolts.datasets.base_dataset.LightDataset
```

Customized CIFAR10 dataset for testing Pytorch Lightning without the torchvision dependency.

Part of the code was copied from <https://github.com/pytorch/vision/blob/build/v0.5.0/torchvision/datasets/>

Parameters

- **data_dir** (str) – Root directory of dataset where CIFAR10/processed/training.pt and CIFAR10/processed/test.pt exist.
- **train** (bool) – If True, creates dataset from training.pt, otherwise from test.pt.
- **download** (bool) – If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.

Examples

```
>>> from torchvision import transforms
>>> from pl_bolts.transforms.dataset_normalizations import cifar10_normalization
>>> cf10_transforms = transforms.Compose([transforms.ToTensor(), cifar10_
↳ normalization()])
>>> dataset = CIFAR10(download=True, transform=cf10_transforms, data_dir="datasets
↳ ")
>>> len(dataset)
50000
```

(continues on next page)

(continued from previous page)

```
>>> torch.bincount(dataset.targets)
tensor([5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000, 5000])
>>> data, label = dataset[0]
>>> data.shape
torch.Size([3, 32, 32])
>>> label
6
```

Labels:

```
airplane: 0
automobile: 1
bird: 2
cat: 3
deer: 4
dog: 5
frog: 6
horse: 7
ship: 8
truck: 9
```

classmethod `_check_exists` (*data_folder*, *file_names*)

Return type `bool`

_extract_archive_save_torch (*download_path*)

_unpickle (*path_folder*, *file_name*)

Return type `Tuple[Tensor, Tensor]`

download (*data_folder*)

Download the data if it doesn't exist in `cached_folder_path` already.

Return type `None`

prepare_data (*download*)

BASE_URL = 'https://www.cs.toronto.edu/~kriz/'

DATASET_NAME = 'CIFAR10'

FILE_NAME = 'cifar-10-python.tar.gz'

TEST_FILE_NAME = 'test.pt'

TRAIN_FILE_NAME = 'training.pt'

cache_folder_name: `str` = 'complete'

data = `None`

dir_path = `None`

labels = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

normalize = `None`

relabel = `False`

targets = `None`

```
class pl_bolts.datasets.cifar10_dataset.TrialCIFAR10 (data_dir='.', train=True, transform=None, download=False, num_samples=100, labels=(1, 5, 8), relabel=True)
```

Bases: `pl_bolts.datasets.cifar10_dataset.CIFAR10`

Customized CIFAR10 dataset for testing Pytorch Lightning without the torchvision dependency. .. rubric:: Examples

```
>>> dataset = TrialCIFAR10(download=True, num_samples=150, labels=(1, 5, 8), data_dir="datasets")
>>> len(dataset)
450
>>> sorted(set([d.item() for d in dataset.targets]))
[1, 5, 8]
>>> torch.bincount(dataset.targets)
tensor([ 0, 150,  0,  0,  0, 150,  0,  0, 150])
>>> data, label = dataset[0]
>>> data.shape
torch.Size([3, 32, 32])
```

Parameters

- **data_dir** (`str`) – Root directory of dataset where CIFAR10/processed/training.pt and CIFAR10/processed/test.pt exist.
- **train** (`bool`) – If True, creates dataset from training.pt, otherwise from test.pt.
- **download** (`bool`) – If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.
- **num_samples** (`int`) – number of examples per selected class/digit
- **labels** (`Optional[Sequence]`) – list selected CIFAR10 digits/classes

prepare_data (*download*)

Return type `None`

data = `None`

dir_path = `None`

normalize = `None`

targets = `None`

`pl_bolts.datasets.concat_dataset` module

```
class pl_bolts.datasets.concat_dataset.ConcatDataset (*datasets)
```

Bases: `torch.utils.data.Dataset`

pl_bolts.datasets.dummy_dataset module

class pl_bolts.datasets.dummy_dataset.**DummyDataset** (**shapes*, *num_samples*=10000)
Bases: torch.utils.data.Dataset

Generate a dummy dataset

Example

```
>>> from pl_bolts.datasets import DummyDataset
>>> from torch.utils.data import DataLoader
>>> # mnist dims
>>> ds = DummyDataset((1, 28, 28), (1, ))
>>> dl = DataLoader(ds, batch_size=7)
>>> # get first batch
>>> batch = next(iter(dl))
>>> x, y = batch
>>> x.size()
torch.Size([7, 1, 28, 28])
>>> y.size()
torch.Size([7, 1])
```

Parameters

- ***shapes** – list of shapes
- **num_samples** (*int*) – how many samples to use in this dataset

class pl_bolts.datasets.dummy_dataset.**DummyDetectionDataset** (*img_shape*=(3, 256, 256), *num_boxes*=1, *num_classes*=2, *num_samples*=10000)
Bases: torch.utils.data.Dataset

Generate a dummy dataset for detection

Example

```
>>> from pl_bolts.datasets import DummyDetectionDataset
>>> from torch.utils.data import DataLoader
>>> ds = DummyDetectionDataset()
>>> dl = DataLoader(ds, batch_size=7)
```

Parameters

- ***shapes** – list of shapes
- **num_samples** (*int*) – how many samples to use in this dataset

_random_bbox ()

class pl_bolts.datasets.dummy_dataset.**RandomDataset** (*size*, *num_samples*=250)
Bases: torch.utils.data.Dataset

Generate a dummy dataset

Example

```
>>> from pl_bolts.datasets import RandomDataset
>>> from torch.utils.data import DataLoader
>>> ds = RandomDataset(10)
>>> dl = DataLoader(ds, batch_size=7)
```

Parameters

- **size** (*int*) – tuple
- **num_samples** (*int*) – number of samples

class `pl_bolts.datasets.dummy_dataset.RandomDictDataset` (*size*, *num_samples=250*)

Bases: `torch.utils.data.Dataset`

Generate a dummy dataset with a dict structure

Example

```
>>> from pl_bolts.datasets import RandomDictDataset
>>> from torch.utils.data import DataLoader
>>> ds = RandomDictDataset(10)
>>> dl = DataLoader(ds, batch_size=7)
```

Parameters

- **size** (*int*) – tuple
- **num_samples** (*int*) – number of samples

class `pl_bolts.datasets.dummy_dataset.RandomDictStringDataset` (*size*,
num_samples=250)

Bases: `torch.utils.data.Dataset`

Generate a dummy dataset with strings

Example

```
>>> from pl_bolts.datasets import RandomDictStringDataset
>>> from torch.utils.data import DataLoader
>>> ds = RandomDictStringDataset(10)
>>> dl = DataLoader(ds, batch_size=7)
```

Parameters

- **size** (*int*) – tuple
- **num_samples** (*int*) – number of samples

pl_bolts.datasets.imagenet_dataset module

```
class pl_bolts.datasets.imagenet_dataset.UnlabeledImagenet (root,      split='train',  
                                                         num_classes=-1,  
                                                         num_imgs_per_class=-  
                                                         1,  
                                                         num_imgs_per_class_val_split=50,  
                                                         meta_dir=None,  
                                                         **kwargs)
```

Bases: `object`

Official train set gets split into train, val. (using `nb_imgs_per_val_class` for each class). Official validation becomes test set

Within each class, we further allow limiting the number of samples per class (for semi-sup lng)

Parameters

- **root** – path of dataset
- **split** (`str`) –
- **num_classes** (`int`) – Sets the limit of classes
- **num_imgs_per_class** (`int`) – Limits the number of images per class
- **num_imgs_per_class_val_split** (`int`) – How many images per class to generate the val split
- **download** –
- **kwargs** –

classmethod `generate_meta_bins` (*devkit_dir*)

partition_train_set (*imgs, nb_imgs_in_val*)

`pl_bolts.datasets.imagenet_dataset._calculate_md5` (*fpath, chunk_size=1048576*)

`pl_bolts.datasets.imagenet_dataset._check_integrity` (*fpath, md5=None*)

`pl_bolts.datasets.imagenet_dataset._check_md5` (*fpath, md5, **kwargs*)

`pl_bolts.datasets.imagenet_dataset._is_gzip` (*filename*)

`pl_bolts.datasets.imagenet_dataset._is_tar` (*filename*)

`pl_bolts.datasets.imagenet_dataset._is_targz` (*filename*)

`pl_bolts.datasets.imagenet_dataset._is_tarxz` (*filename*)

`pl_bolts.datasets.imagenet_dataset._is_zip` (*filename*)

`pl_bolts.datasets.imagenet_dataset._verify_archive` (*root, file, md5*)

`pl_bolts.datasets.imagenet_dataset.extract_archive` (*from_path, to_path=None, remove_finished=False*)

`pl_bolts.datasets.imagenet_dataset.parse_devkit_archive` (*root, file=None*)

Parse the devkit archive of the ImageNet2012 classification dataset and save the meta information in a binary file.

Parameters

- **root** (`str`) – Root directory containing the devkit archive

- **file** (*str*, *optional*) – Name of devkit archive. Defaults to 'ILSVRC2012_devkit_t12.tar.gz'

pl_bolts.datasets.kitti_dataset module

```
class pl_bolts.datasets.kitti_dataset.KittiDataset (data_dir, img_size=(1242, 376),
                                                void_labels=(0, 1, 2, 3, 4, 5, 6,
9, 10, 14, 15, 16, 18, 29, 30, -1),
                                                valid_labels=(7, 8, 11, 12, 13, 17,
19, 20, 21, 22, 23, 24, 25, 26, 27,
28, 31, 32, 33), transform=None)
```

Bases: `torch.utils.data.Dataset`

Note: You need to have downloaded the Kitti dataset first and provide the path to where it is saved. You can download the dataset here: http://www.cvlibs.net/datasets/kitti/eval_semseg.php?benchmark=semantics2015

There are 34 classes, however not all of them are useful for training (e.g. railings on highways). These useless classes (the pixel values of these classes) are stored in *void_labels*. Useful classes are stored in *valid_labels*.

The *encode_segmap* function sets all pixels with any of the *void_labels* to *ignore_index* (250 by default). It also sets all of the valid pixels to the appropriate value between 0 and *len(valid_labels)* (since that is the number of valid classes), so it can be used properly by the loss function when comparing with the output.

Parameters

- **data_dir** (*str*) – where to load the data from path, i.e. '/path/to/folder/with/data_semantics/'
- **img_size** (*tuple*) – image dimensions (width, height)
- **void_labels** (*list*) – useless classes to be excluded from training
- **valid_labels** (*list*) – useful classes to include

encode_segmap (*mask*)

Sets void classes to zero so they won't be considered for training

get_filenames (*path*)

Returns a list of absolute paths to images inside given *path*

IMAGE_PATH = 'training/image_2'

MASK_PATH = 'training/semantic'

pl_bolts.datasets.mnist_dataset module

```
class pl_bolts.datasets.mnist_dataset.BinaryMNIST
```

Bases: `object`

pl_bolts.datasets.ssl_amdim_datasets module

```
class pl_bolts.datasets.ssl_amdim_datasets.CIFAR10Mixed(root, split='val', trans-  
                                                    form=None, tar-  
                                                    get_transform=None,  
                                                    download=False,  
                                                    nb_labeled_per_class=None,  
                                                    val_pct=0.1)  
    Bases: pl_bolts.datasets.ssl_amdim_datasets.SSLDatasetMixin, object  
  
class pl_bolts.datasets.ssl_amdim_datasets.SSLDatasetMixin  
    Bases: abc.ABC  
  
    classmethod deterministic_shuffle(x, y)  
  
    classmethod generate_train_val_split(examples, labels, pct_val)  
        Splits dataset uniformly across classes  
  
    classmethod select_nb_imgs_per_class(examples, labels, nb_imgs_in_val)  
        Splits a dataset into two parts. The labeled split has nb_imgs_in_val per class
```

32.4 pl_bolts.metrics package

```
pl_bolts.metrics.accuracy(preds, labels)  
pl_bolts.metrics.mean(res, key)  
pl_bolts.metrics.precision_at_k(output, target, top_k=(1, ))  
    Computes the accuracy over the k top predictions for the specified values of k
```

32.4.1 Submodules

pl_bolts.metrics.aggregation module

```
pl_bolts.metrics.aggregation.accuracy(preds, labels)  
pl_bolts.metrics.aggregation.mean(res, key)  
pl_bolts.metrics.aggregation.precision_at_k(output, target, top_k=(1, ))  
    Computes the accuracy over the k top predictions for the specified values of k
```

pl_bolts.metrics.object_detection module

```
pl_bolts.metrics.object_detection.giou(preds, target)  
    Calculates the generalized intersection over union.
```

It has been proposed in [Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression](#).

Parameters

- **preds** (*Tensor*) – an Nx4 batch of prediction bounding boxes with representation [x_min, y_min, x_max, y_max]
- **target** (*Tensor*) – an Mx4 batch of target bounding boxes with representation [x_min, y_min, x_max, y_max]

Example

```
>>> import torch
>>> from pl_bolts.metrics.object_detection import giou
>>> preds = torch.tensor([[100, 100, 200, 200]])
>>> target = torch.tensor([[150, 150, 250, 250]])
>>> giou(preds, target)
tensor([[ -0.0794]])
```

Return type `Tensor`

Returns GIoU in an NxM tensor containing the pairwise GIoU values for every element in preds and target, where N is the number of prediction bounding boxes and M is the number of target bounding boxes

`pl_bolts.metrics.object_detection.iou(preds, target)`
Calculates the intersection over union.

Parameters

- **preds** (`Tensor`) – an Nx4 batch of prediction bounding boxes with representation [x_min, y_min, x_max, y_max]
- **target** (`Tensor`) – an Mx4 batch of target bounding boxes with representation [x_min, y_min, x_max, y_max]

Example

```
>>> import torch
>>> from pl_bolts.metrics.object_detection import iou
>>> preds = torch.tensor([[100, 100, 200, 200]])
>>> target = torch.tensor([[150, 150, 250, 250]])
>>> iou(preds, target)
tensor([[ 0.1429]])
```

Returns

an NxM tensor containing the pairwise IoU values for every element in preds and target, where N is the number of prediction bounding boxes and M is the number of target bounding boxes

Return type IoU tensor

32.5 pl_bolts.models package

Collection of PyTorchLightning models

class `pl_bolts.models.AE` (*input_height*, *enc_type*='resnet18', *first_conv*=False, *maxpool1*=False, *enc_out_dim*=512, *latent_dim*=256, *lr*=0.0001, ***kwargs*)
Bases: `pytorch_lightning.LightningModule`

Standard AE

Model is available pretrained on different datasets:

Example:

```
# not pretrained
ae = AE()

# pretrained on cifar10
ae = AE.from_pretrained('cifar10-resnet18')
```

Parameters

- **input_height** (*int*) – height of the images
- **enc_type** (*str*) – option between resnet18 or resnet50
- **first_conv** (*bool*) – use standard kernel_size 7, stride 2 at start or replace it with kernel_size 3, stride 1 conv
- **maxpool1** (*bool*) – use standard maxpool to reduce spatial dim of feat by a factor of 2
- **enc_out_dim** (*int*) – set according to the out_channel count of encoder used (512 for resnet18, 2048 for resnet50)
- **latent_dim** (*int*) – dim of latent space
- **lr** (*float*) – learning rate for Adam

```
static add_model_specific_args (parent_parser)
```

```
configure_optimizers ()
```

```
forward (x)
```

```
from_pretrained (checkpoint_name)
```

```
static pretrained_weights_available ()
```

```
step (batch, batch_idx)
```

```
training_step (batch, batch_idx)
```

```
validation_step (batch, batch_idx)
```

```
pretrained_urls = {'cifar10-resnet18': 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/cifar10-resnet18-weights.tar.gz'}
```

```
class pl_bolts.models.VAE (input_height, enc_type='resnet18', first_conv=False, maxpool1=False,
                           enc_out_dim=512, kl_coeff=0.1, latent_dim=256, lr=0.0001,
                           **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Standard VAE with Gaussian Prior and approx posterior.

Model is available pretrained on different datasets:

Example:

```
# not pretrained
vae = VAE ()

# pretrained on cifar10
vae = VAE.from_pretrained('cifar10-resnet18')

# pretrained on stl10
vae = VAE.from_pretrained('stl10-resnet18')
```

Parameters

- **input_height** (*int*) – height of the images
- **enc_type** (*str*) – option between resnet18 or resnet50
- **first_conv** (*bool*) – use standard kernel_size 7, stride 2 at start or replace it with kernel_size 3, stride 1 conv
- **maxpool1** (*bool*) – use standard maxpool to reduce spatial dim of feat by a factor of 2
- **enc_out_dim** (*int*) – set according to the out_channel count of encoder used (512 for resnet18, 2048 for resnet50)
- **kl_coeff** (*float*) – coefficient for kl term of the loss
- **latent_dim** (*int*) – dim of latent space
- **lr** (*float*) – learning rate for Adam

```

_run_step(x)

static add_model_specific_args(parent_parser)

configure_optimizers()

forward(x)

from_pretrained(checkpoint_name)

static pretrained_weights_available()

sample(mu, log_var)

step(batch, batch_idx)

training_step(batch, batch_idx)

validation_step(batch, batch_idx)

pretrained_urls = {'cifar10-resnet18': 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com'}

class pl_bolts.models.LitMNIST(hidden_dim=128, learning_rate=0.001, batch_size=32,
                               num_workers=4, data_dir="", **kwargs)
    Bases: pytorch_lightning.LightningModule

    static add_model_specific_args(parent_parser)

    configure_optimizers()

    forward(x)

    prepare_data()

    test_dataloader()

    test_step(batch, batch_idx)

    train_dataloader()

    training_step(batch, batch_idx)

    val_dataloader()

    validation_step(batch, batch_idx)

class pl_bolts.models.LinearRegression(input_dim, output_dim=1, bias=True, learning_rate=0.0001,
                                       optimizer=torch.optim.Adam, l1_strength=0.0, l2_strength=0.0, **kwargs)
    Bases: pytorch_lightning.LightningModule

```

Linear regression model implementing - with optional L1/L2 regularization $\min_{\{W\}} \| (Wx + b) - y \|_2^2$

Parameters

- **input_dim** (`int`) – number of dimensions of the input (1+)
- **output_dim** (`int`) – number of dimensions of the output (default=1)
- **bias** (`bool`) – If false, will not use $+b$
- **learning_rate** (`float`) – learning_rate for the optimizer
- **optimizer** (`Optimizer`) – the optimizer to use (default='Adam')
- **l1_strength** (`float`) – L1 regularization strength (default=None)
- **l2_strength** (`float`) – L2 regularization strength (default=None)

```
static add_model_specific_args (parent_parser)
```

```
configure_optimizers ()
```

```
forward (x)
```

```
test_epoch_end (outputs)
```

```
test_step (batch, batch_idx)
```

```
training_step (batch, batch_idx)
```

```
validation_epoch_end (outputs)
```

```
validation_step (batch, batch_idx)
```

```
class pl_bolts.models.LogisticRegression (input_dim, num_classes, bias=True, learning_rate=0.0001, optimizer=torch.optim.Adam, l1_strength=0.0, l2_strength=0.0, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Logistic regression model

Parameters

- **input_dim** (`int`) – number of dimensions of the input (at least 1)
- **num_classes** (`int`) – number of class labels (binary: 2, multi-class: >2)
- **bias** (`bool`) – specifies if a constant or intercept should be fitted (equivalent to `fit_intercept` in `sklearn`)
- **learning_rate** (`float`) – learning_rate for the optimizer
- **optimizer** (`Optimizer`) – the optimizer to use (default='Adam')
- **l1_strength** (`float`) – L1 regularization strength (default=None)
- **l2_strength** (`float`) – L2 regularization strength (default=None)

```
static add_model_specific_args (parent_parser)
```

```
configure_optimizers ()
```

```
forward (x)
```

```
test_epoch_end (outputs)
```

```
test_step (batch, batch_idx)
```

```
training_step (batch, batch_idx)
```


validation_epoch_end (*outputs*)

validation_step (*batch, batch_idx*)

class `pl_bolts.models.PixelCNN` (*input_channels, hidden_channels=256, num_blocks=5*)

Bases: `torch.nn.Module`

Implementation of `Pixel CNN`.

Paper authors: Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, Koray Kavukcuoglu

Implemented by:

- William Falcon

Example:

```
>>> from pl_bolts.models.vision import PixelCNN
>>> import torch
...
>>> model = PixelCNN(input_channels=3)
>>> x = torch.rand(5, 3, 64, 64)
>>> out = model(x)
...
>>> out.shape
torch.Size([5, 3, 64, 64])
```

conv_block (*input_channels*)

forward (*z*)

class `pl_bolts.models.SemSegment` (*lr=0.01, num_classes=19, num_layers=5, features_start=64, bilinear=False*)

Bases: `pytorch_lightning.LightningModule`

Basic model for semantic segmentation. Uses UNet architecture by default.

The default parameters in this model are for the KITTI dataset. Note, if you'd like to use this model as is, you will first need to download the KITTI dataset yourself. You can download the dataset [here](#).

Implemented by:

- Annika Brundyn

Parameters

- **num_layers** (*int*) – number of layers in each side of U-net (default 5)
- **features_start** (*int*) – number of features in first layer (default 64)
- **bilinear** (*bool*) – whether to use bilinear interpolation (True) or transposed convolutions (default) for upsampling.
- **lr** (*float*) – learning (default 0.01)

static add_model_specific_args (*parent_parser*)

configure_optimizers ()

forward (*x*)

training_step (*batch, batch_nb*)

validation_epoch_end (*outputs*)

validation_step (*batch, batch_idx*)

class `pl_bolts.models.UNet` (*num_classes, input_channels=3, num_layers=5, features_start=64, bilinear=False*)

Bases: `torch.nn.Module`

Paper: [U-Net: Convolutional Networks for Biomedical Image Segmentation](#)

Paper authors: Olaf Ronneberger, Philipp Fischer, Thomas Brox

Implemented by:

- [Annika Brundyn](#)
- [Akshay Kulkarni](#)

Parameters

- **num_classes** (*int*) – Number of output classes required
- **input_channels** (*int*) – Number of channels in input images (default 3)
- **num_layers** (*int*) – Number of layers in each side of U-net (default 5)
- **features_start** (*int*) – Number of features in first layer (default 64)
- **bilinear** (*bool*) – Whether to use bilinear interpolation or transposed convolutions (default) for upsampling.

forward (*x*)

class `pl_bolts.models.GPT2` (*embed_dim, heads, layers, num_positions, vocab_size, num_classes*)

Bases: `pytorch_lightning.LightningModule`

GPT-2 from [language Models are Unsupervised Multitask Learners](#)

Paper by: Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever

Implementation contributed by:

- [Teddy Koker](#)

Example:

```
from pl_bolts.models import GPT2

seq_len = 17
batch_size = 32
vocab_size = 16
x = torch.randint(0, vocab_size, (seq_len, batch_size))
model = GPT2(embed_dim=32, heads=2, layers=2, num_positions=seq_len, vocab_
↪size=vocab_size, num_classes=4)
results = model(x)
```

_init_embeddings ()

_init_layers ()

_init_sos_token ()

forward (*x, classify=False*)

Expect input as shape [sequence len, batch] If classify, return classification logits

```
class pl_bolts.models.ImageGPT(embed_dim=16, heads=2, layers=2, pixels=28, vocab_size=16,
                               num_classes=10, classify=False, batch_size=64, learn-
                               ing_rate=0.01, steps=25000, data_dir='.', num_workers=8,
                               **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Paper: [Generative Pretraining from Pixels](#) [original paper [code](#)].

Paper by: Mark Che, Alec Radford, Rewon Child, Jeff Wu, Heewoo Jun, Prafulla Dhariwal, David Luan, Ilya Sutskever

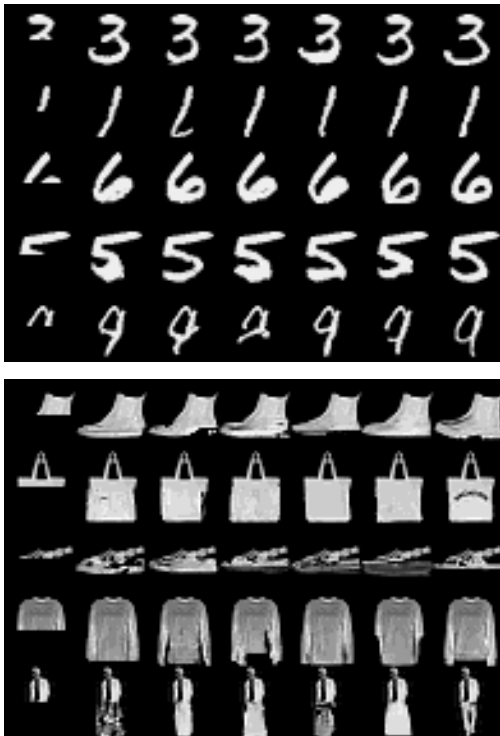
Implementation contributed by:

- [Teddy Koker](#)

Original repo with results and more implementation details:

- <https://github.com/teddykoker/image-gpt>

Example Results (Photo credits: Teddy Koker):



Default arguments:

Table 1: Argument Defaults

Argument	Default	iGPT-S (Chen et al.)
<code>-embed_dim</code>	16	512
<code>-heads</code>	2	8
<code>-layers</code>	8	24
<code>-pixels</code>	28	32
<code>-vocab_size</code>	16	512
<code>-num_classes</code>	10	10
<code>-batch_size</code>	64	128
<code>-learning_rate</code>	0.01	0.01
<code>-steps</code>	25000	1000000

Example:

```
import pytorch_lightning as pl
from pl_bolts.models.vision import ImageGPT

dm = MNISTDataModule('.')
model = ImageGPT(dm)

pl.Trainer(gpu=4).fit(model)
```

As script:

```
cd pl_bolts/models/vision/image_gpt
python igpt_module.py --learning_rate 1e-2 --batch_size 32 --gpus 4
```

Parameters

- **embed_dim** (*int*) – the embedding dim
- **heads** (*int*) – number of attention heads
- **layers** (*int*) – number of layers
- **pixels** (*int*) – number of input pixels
- **vocab_size** (*int*) – vocab size
- **num_classes** (*int*) – number of classes in the input
- **classify** (*bool*) – true if should classify
- **batch_size** (*int*) – the batch size
- **learning_rate** (*float*) – learning rate
- **steps** (*int*) – number of steps for cosine annealing
- **data_dir** (*str*) – where to store data
- **num_workers** (*int*) – num_data workers

static add_model_specific_args (*parent_parser*)

configure_optimizers ()

forward (*x*, *classify=False*)

test_epoch_end (*outs*)

test_step (*batch*, *batch_idx*)

training_step (*batch*, *batch_idx*)

validation_epoch_end (*outs*)

validation_step (*batch*, *batch_idx*)

32.5.1 Subpackages

pl_bolts.models.autoencoders package

Here are a VAE and GAN

```
class pl_bolts.models.autoencoders.AE(input_height, enc_type='resnet18', first_conv=False,
                                       maxpool1=False, enc_out_dim=512, latent_dim=256, lr=0.0001, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Standard AE

Model is available pretrained on different datasets:

Example:

```
# not pretrained
ae = AE()

# pretrained on cifar10
ae = AE.from_pretrained('cifar10-resnet18')
```

Parameters

- **input_height** (`int`) – height of the images
- **enc_type** (`str`) – option between resnet18 or resnet50
- **first_conv** (`bool`) – use standard kernel_size 7, stride 2 at start or replace it with kernel_size 3, stride 1 conv
- **maxpool1** (`bool`) – use standard maxpool to reduce spatial dim of feat by a factor of 2
- **enc_out_dim** (`int`) – set according to the out_channel count of encoder used (512 for resnet18, 2048 for resnet50)
- **latent_dim** (`int`) – dim of latent space
- **lr** (`float`) – learning rate for Adam

static add_model_specific_args (`parent_parser`)

configure_optimizers ()

forward (`x`)

from_pretrained (`checkpoint_name`)

static pretrained_weights_available ()

step (`batch`, `batch_idx`)

training_step (`batch`, `batch_idx`)

validation_step (`batch`, `batch_idx`)

pretrained_urls = {'cifar10-resnet18': 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/cifar10-resnet18-weights.tar.gz'}

```
class pl_bolts.models.autoencoders.VAE(input_height, enc_type='resnet18', first_conv=False,
                                       maxpool1=False, enc_out_dim=512, kl_coeff=0.1,
                                       latent_dim=256, lr=0.0001, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Standard VAE with Gaussian Prior and approx posterior.

Model is available pretrained on different datasets:

Example:

```
# not pretrained
vae = VAE()

# pretrained on cifar10
vae = VAE.from_pretrained('cifar10-resnet18')

# pretrained on stl10
vae = VAE.from_pretrained('stl10-resnet18')
```

Parameters

- **input_height** (*int*) – height of the images
- **enc_type** (*str*) – option between resnet18 or resnet50
- **first_conv** (*bool*) – use standard kernel_size 7, stride 2 at start or replace it with kernel_size 3, stride 1 conv
- **maxpool1** (*bool*) – use standard maxpool to reduce spatial dim of feat by a factor of 2
- **enc_out_dim** (*int*) – set according to the out_channel count of encoder used (512 for resnet18, 2048 for resnet50)
- **kl_coeff** (*float*) – coefficient for kl term of the loss
- **latent_dim** (*int*) – dim of latent space
- **lr** (*float*) – learning rate for Adam

_run_step (*x*)

static add_model_specific_args (*parent_parser*)

configure_optimizers ()

forward (*x*)

from_pretrained (*checkpoint_name*)

static pretrained_weights_available ()

sample (*mu*, *log_var*)

step (*batch*, *batch_idx*)

training_step (*batch*, *batch_idx*)

validation_step (*batch*, *batch_idx*)

pretrained_urls = {'cifar10-resnet18': 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/cifar10-resnet18-weights.tar.gz'}

`pl_bolts.models.autoencoders.resnet18_decoder` (*latent_dim*, *input_height*, *first_conv*, *maxpool1*)

`pl_bolts.models.autoencoders.resnet18_encoder` (*first_conv*, *maxpool1*)

`pl_bolts.models.autoencoders.resnet50_decoder` (*latent_dim*, *input_height*, *first_conv*, *maxpool1*)

`pl_bolts.models.autoencoders.resnet50_encoder` (*first_conv*, *maxpool1*)

Subpackages

pl_bolts.models.autoencoders.basic_ae package

AE Template

This is a basic template for implementing an Autoencoder in PyTorch Lightning.

A default encoder and decoder have been provided but can easily be replaced by custom models.

This template uses the CIFAR10 dataset but image data of any dimension can be fed in as long as the image width and image height are even values. For other types of data, such as sound, it will be necessary to change the Encoder and Decoder.

The default encoder is a resnet18 backbone followed by linear layers which map representations to latent space. The default decoder mirrors the encoder architecture and is similar to an inverted resnet18.

```
from pl_bolts.models.autoencoders import AE

model = AE()
trainer = pl.Trainer()
trainer.fit(model)
```

Submodules

pl_bolts.models.autoencoders.basic_ae.basic_ae_module module

```
class pl_bolts.models.autoencoders.basic_ae.basic_ae_module.AE(input_height,
                                                                enc_type='resnet18',
                                                                first_conv=False,
                                                                max-
                                                                pool1=False,
                                                                enc_out_dim=512,
                                                                la-
                                                                tent_dim=256,
                                                                lr=0.0001,
                                                                **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Standard AE

Model is available pretrained on different datasets:

Example:

```
# not pretrained
ae = AE()

# pretrained on cifar10
ae = AE.from_pretrained('cifar10-resnet18')
```

Parameters

- **input_height** (`int`) – height of the images
- **enc_type** (`str`) – option between resnet18 or resnet50

- **first_conv** (*bool*) – use standard kernel_size 7, stride 2 at start or replace it with kernel_size 3, stride 1 conv
- **maxpool1** (*bool*) – use standard maxpool to reduce spatial dim of feat by a factor of 2
- **enc_out_dim** (*int*) – set according to the out_channel count of encoder used (512 for resnet18, 2048 for resnet50)
- **latent_dim** (*int*) – dim of latent space
- **lr** (*float*) – learning rate for Adam

```
static add_model_specific_args (parent_parser)

configure_optimizers ()

forward (x)

from_pretrained (checkpoint_name)

static pretrained_weights_available ()

step (batch, batch_idx)

training_step (batch, batch_idx)

validation_step (batch, batch_idx)

pretrained_urls = {'cifar10-resnet18': 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/cifar10-resnet18.pth'}

pl_bolts.models.autoencoders.basic_ae.basic_ae_module.cli_main (args=None)
```

pl_bolts.models.autoencoders.basic_vae package

VAE Template

This is a basic template for implementing a Variational Autoencoder in PyTorch Lightning.

A default encoder and decoder have been provided but can easily be replaced by custom models.

This template uses the CIFAR10 dataset but image data of any dimension can be fed in as long as the image width and image height are even values. For other types of data, such as sound, it will be necessary to change the Encoder and Decoder.

The default encoder is a resnet18 backbone followed by linear layers which map representations to μ and σ . The default decoder mirrors the encoder architecture and is similar to an inverted resnet18. The model also assumes a Gaussian prior and a Gaussian approximate posterior distribution.

Submodules

pl_bolts.models.autoencoders.basic_vae.basic_vae_module module

```
class pl_bolts.models.autoencoders.basic_vae.basic_vae_module.VAE(input_height,
                                                                    enc_type='resnet18',
                                                                    first_conv=False,
                                                                    max-
                                                                    pool1=False,
                                                                    enc_out_dim=512,
                                                                    kl_coeff=0.1,
                                                                    la-
                                                                    tent_dim=256,
                                                                    lr=0.0001,
                                                                    **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Standard VAE with Gaussian Prior and approx posterior.

Model is available pretrained on different datasets:

Example:

```
# not pretrained
vae = VAE()

# pretrained on cifar10
vae = VAE.from_pretrained('cifar10-resnet18')

# pretrained on stl10
vae = VAE.from_pretrained('stl10-resnet18')
```

Parameters

- **input_height** (`int`) – height of the images
- **enc_type** (`str`) – option between resnet18 or resnet50
- **first_conv** (`bool`) – use standard kernel_size 7, stride 2 at start or replace it with kernel_size 3, stride 1 conv
- **maxpool1** (`bool`) – use standard maxpool to reduce spatial dim of feat by a factor of 2
- **enc_out_dim** (`int`) – set according to the out_channel count of encoder used (512 for resnet18, 2048 for resnet50)
- **kl_coeff** (`float`) – coefficient for kl term of the loss
- **latent_dim** (`int`) – dim of latent space
- **lr** (`float`) – learning rate for Adam

_run_step (`x`)

static add_model_specific_args (`parent_parser`)

configure_optimizers ()

forward (`x`)

from_pretrained (`checkpoint_name`)

static pretrained_weights_available ()

sample (`mu`, `log_var`)

```
step(batch, batch_idx)
training_step(batch, batch_idx)
validation_step(batch, batch_idx)
pretrained_urls = {'cifar10-resnet18': 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/cifar10-resnet18.pth'}
pl_bolts.models.autoencoders.basic_vae.basic_vae_module.cli_main(args=None)
```

Submodules

pl_bolts.models.autoencoders.components module

```
class pl_bolts.models.autoencoders.components.DecoderBlock(inplanes, planes,
                                                            scale=1, upsam-
                                                            ple=None)
```

Bases: `torch.nn.Module`

ResNet block, but convs replaced with resize convs, and channel increase is in second conv, not first

forward(*x*)

expansion = 1

```
class pl_bolts.models.autoencoders.components.DecoderBottleneck(inplanes, planes,
                                                                  scale=1, upsam-
                                                                  ple=None)
```

Bases: `torch.nn.Module`

ResNet bottleneck, but convs replaced with resize convs

forward(*x*)

expansion = 4

```
class pl_bolts.models.autoencoders.components.EncoderBlock(inplanes, planes,
                                                            stride=1, downsam-
                                                            ple=None)
```

Bases: `torch.nn.Module`

ResNet block, copied from <https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py#L35>

forward(*x*)

expansion = 1

```
class pl_bolts.models.autoencoders.components.EncoderBottleneck(inplanes, planes,
                                                                  stride=1, downsam-
                                                                  ple=None)
```

Bases: `torch.nn.Module`

ResNet bottleneck, copied from <https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py#L75>

forward(*x*)

expansion = 4

```

class pl_bolts.models.autoencoders.components.Interpolate (size=None,
                                                            scale_factor=None)
    Bases: torch.nn.Module
    nn.Module wrapper for F.interpolate
    forward (x)

class pl_bolts.models.autoencoders.components.ResNetDecoder (block,          layers,
                                                             latent_dim,          in-
                                                             put_height,
                                                             first_conv=False,
                                                             maxpool1=False)
    Bases: torch.nn.Module
    Resnet in reverse order
    _make_layer (block, planes, blocks, scale=1)
    forward (x)

class pl_bolts.models.autoencoders.components.ResNetEncoder (block,          layers,
                                                             first_conv=False,
                                                             maxpool1=False)
    Bases: torch.nn.Module
    _make_layer (block, planes, blocks, stride=1)
    forward (x)

pl_bolts.models.autoencoders.components.conv1x1 (in_planes, out_planes, stride=1)
    1x1 convolution

pl_bolts.models.autoencoders.components.conv3x3 (in_planes, out_planes, stride=1)
    3x3 convolution with padding

pl_bolts.models.autoencoders.components.resize_conv1x1 (in_planes,          out_planes,
                                                         scale=1)
    upsample + 1x1 convolution with padding to avoid checkerboard artifact

pl_bolts.models.autoencoders.components.resize_conv3x3 (in_planes,          out_planes,
                                                         scale=1)
    upsample + 3x3 convolution with padding to avoid checkerboard artifact

pl_bolts.models.autoencoders.components.resnet18_decoder (latent_dim, input_height,
                                                         first_conv, maxpool1)

pl_bolts.models.autoencoders.components.resnet18_encoder (first_conv, maxpool1)

pl_bolts.models.autoencoders.components.resnet50_decoder (latent_dim, input_height,
                                                         first_conv, maxpool1)

pl_bolts.models.autoencoders.components.resnet50_encoder (first_conv, maxpool1)

```

pl_bolts.models.detection package

```
class pl_bolts.models.detection.FasterRCNN (learning_rate=0.0001, num_classes=91,  
                                             backbone=None, fpn=True, pretrained=False,  
                                             pretrained_backbone=True, trainable_backbone_layers=3, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks](#).

Paper authors: Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun

Model implemented by:

- *Teddy Koker* <<https://github.com/teddykoker>>

During training, the model expects both the input tensors, as well as targets (list of dictionary), containing:

- `boxes` (`FloatTensor[N, 4]`): the ground truth boxes in `[x1, y1, x2, y2]` format.
- `labels` (`Int64Tensor[N]`): the class label for each ground truth box

CLI command:

```
# PascalVOC  
python faster_rcnn.py --gpus 1 --pretrained True
```

Parameters

- **learning_rate** (`float`) – the learning rate
- **num_classes** (`int`) – number of detection classes (including background)
- **backbone** (`Optional[str]`) – Pretained backbone CNN architecture.
- **fpn** (`bool`) – If True, creates a Feature Pyramid Network on top of Resnet based CNNs.
- **pretrained** (`bool`) – if true, returns a model pre-trained on COCO train2017
- **pretrained_backbone** (`bool`) – if true, returns a model with backbone pre-trained on Imagenet
- **trainable_backbone_layers** (`int`) – number of trainable resnet layers starting from final block

static `add_model_specific_args` (*parent_parser*)

`configure_optimizers` ()

`forward` (*x*)

`training_step` (*batch, batch_idx*)

`validation_epoch_end` (*outs*)

`validation_step` (*batch, batch_idx*)

Subpackages

pl_bolts.models.detection.components package

`pl_bolts.models.detection.components.create_torchvision_backbone` (*model_name*,
pre-trained=True)

Creates CNN backbone from Torchvision.

Parameters

- **model_name** (*str*) – Name of the model. E.g. resnet18
- **pretrained** (*bool*) – Pretrained weights dataset “imagenet”, etc

Return type `Tuple[Module, int]`

Submodules

pl_bolts.models.detection.components._supported_models module

pl_bolts.models.detection.components.torchvision_backbones module

`pl_bolts.models.detection.components.torchvision_backbones._create_backbone_adaptive` (*model*,
out_channels)

Creates backbone by removing linear after Adaptive Pooling layer.

Parameters

- **model** (*Module*) – torch.nn model with adaptive pooling layer
- **out_channels** (*Optional[int]*) – Number of out_channels in last layer

Return type `Module`

`pl_bolts.models.detection.components.torchvision_backbones._create_backbone_features` (*model*,
out_channels)

Creates backbone from feature sequential block.

Parameters

- **model** (*Module*) – torch.nn model with features as sequential block.
- **out_channels** (*int*) – Number of out_channels in last layer.

Return type `Module`

`pl_bolts.models.detection.components.torchvision_backbones._create_backbone_generic` (*model*,
out_channels)

Generic Backbone creator. It removes the last linear layer.

Parameters

- **model** (*Module*) – torch.nn model
- **out_channels** (*int*) – Number of out_channels in last layer.

Return type `Module`

`pl_bolts.models.detection.components.torchvision_backbones.create_torchvision_backbone` (*model, pretrained*)

Creates CNN backbone from Torchvision.

Parameters

- **model_name** (`str`) – Name of the model. E.g. resnet18
- **pretrained** (`bool`) – Pretrained weights dataset “imagenet”, etc

Return type `Tuple[Module, int]`

`pl_bolts.models.detection.faster_rcnn` package

```
class pl_bolts.models.detection.faster_rcnn.FasterRCNN (learning_rate=0.0001,
                                                         num_classes=91,      back-
                                                         bone=None,      fpn=True,
                                                         pretrained=False,  pre-
                                                         trained_backbone=True,
                                                         train-
                                                         able_backbone_layers=3,
                                                         **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks](#).

Paper authors: Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun

Model implemented by:

- *Teddy Koker* <<https://github.com/teddykoker>>

During training, the model expects both the input tensors, as well as targets (list of dictionary), containing:

- **boxes** (`FloatTensor[N, 4]`): the ground truth boxes in `[x1, y1, x2, y2]` format.
- **labels** (`Int64Tensor[N]`): the class label for each ground truth box

CLI command:

```
# PascalVOC
python faster_rcnn.py --gpus 1 --pretrained True
```

Parameters

- **learning_rate** (`float`) – the learning rate
- **num_classes** (`int`) – number of detection classes (including background)
- **backbone** (`Optional[str]`) – Pretained backbone CNN architecture.
- **fpn** (`bool`) – If True, creates a Feature Pyramid Network on top of Resnet based CNNs.
- **pretrained** (`bool`) – if true, returns a model pre-trained on COCO train2017
- **pretrained_backbone** (`bool`) – if true, returns a model with backbone pre-trained on Imagenet
- **trainable_backbone_layers** (`int`) – number of trainable resnet layers starting from final block

```

static add_model_specific_args (parent_parser)

configure_optimizers ()

forward (x)

training_step (batch, batch_idx)

validation_epoch_end (outs)

validation_step (batch, batch_idx)

pl_bolts.models.detection.faster_rcnn.create_fasterrcnn_backbone (backbone,
                                                                    fpn=True,
                                                                    pre-
                                                                    trained=None,
                                                                    train-
                                                                    able_backbone_layers=3,
                                                                    **kwargs)

```

Parameters

- **backbone** (`str`) – Supported backbones are: “resnet18”, “resnet34”, “resnet50”, “resnet101”, “resnet152”, “resnext50_32x4d”, “resnext101_32x8d”, “wide_resnet50_2”, “wide_resnet101_2”, as resnets with fpn backbones. Without fpn backbones supported are: “resnet18”, “resnet34”, “resnet50”, “resnet101”, “resnet152”, “resnext101_32x8d”, “mobilenet_v2”, “vgg11”, “vgg13”, “vgg16”, “vgg19”,
- **fpn** (`bool`) – If True then constructs fpn as well.
- **pretrained** (`Optional[str]`) – If None creates imagenet weights backbone.
- **trainable_backbone_layers** (`int`) – number of trainable resnet layers starting from final block.

Return type `Module`

Submodules

`pl_bolts.models.detection.faster_rcnn.backbones` module

```

pl_bolts.models.detection.faster_rcnn.backbones.create_fasterrcnn_backbone (backbone,
                                                                                fpn=True,
                                                                                pre-
                                                                                trained=None,
                                                                                train-
                                                                                able_backbone_layers=
                                                                                **kwargs)

```

Parameters

- **backbone** (`str`) – Supported backbones are: “resnet18”, “resnet34”, “resnet50”, “resnet101”, “resnet152”, “resnext50_32x4d”, “resnext101_32x8d”, “wide_resnet50_2”, “wide_resnet101_2”, as resnets with fpn backbones. Without fpn backbones supported are: “resnet18”, “resnet34”, “resnet50”, “resnet101”, “resnet152”, “resnext101_32x8d”, “mobilenet_v2”, “vgg11”, “vgg13”, “vgg16”, “vgg19”,
- **fpn** (`bool`) – If True then constructs fpn as well.
- **pretrained** (`Optional[str]`) – If None creates imagenet weights backbone.

- **trainable_backbone_layers** (`int`) – number of trainable resnet layers starting from final block.

Return type `Module`

`pl_bolts.models.detection.faster_rcnn.faster_rcnn_module` module

```
class pl_bolts.models.detection.faster_rcnn.faster_rcnn_module.FasterRCNN (learning_rate=0.0001,  
                                                                    num_classes=91,  
                                                                    back-  
                                                                    bone=None,  
                                                                    fpn=True,  
                                                                    pre-  
                                                                    trained=False,  
                                                                    pre-  
                                                                    trained_backbone=True,  
                                                                    train-  
                                                                    able_backbone_layers=,  
                                                                    **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks](#).

Paper authors: Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun

Model implemented by:

- *Teddy Koker* <<https://github.com/teddykoker>>

During training, the model expects both the input tensors, as well as targets (list of dictionary), containing:

- `boxes` (`FloatTensor[N, 4]`): the ground truth boxes in `[x1, y1, x2, y2]` format.
- `labels` (`Int64Tensor[N]`): the class label for each ground truth box

CLI command:

```
# PascalVOC  
python faster_rcnn.py --gpus 1 --pretrained True
```

Parameters

- **learning_rate** (`float`) – the learning rate
- **num_classes** (`int`) – number of detection classes (including background)
- **backbone** (`Optional[str]`) – Pretained backbone CNN architecture.
- **fpn** (`bool`) – If True, creates a Feature Pyramid Network on top of Resnet based CNNs.
- **pretrained** (`bool`) – if true, returns a model pre-trained on COCO train2017
- **pretrained_backbone** (`bool`) – if true, returns a model with backbone pre-trained on Imagenet
- **trainable_backbone_layers** (`int`) – number of trainable resnet layers starting from final block

static `add_model_specific_args` (`parent_parser`)


```

configure_optimizers()
forward(x)
training_step(batch, batch_idx)
validation_epoch_end(outs)
validation_step(batch, batch_idx)

```

```
pl_bolts.models.detection.faster_rcnn.faster_rcnn_module._evaluate_iou(target,
                                                                    pred)
```

Evaluate intersection over union (IOU) for target from dataset and output prediction from model

```
pl_bolts.models.detection.faster_rcnn.faster_rcnn_module.run_cli()
```

pl_bolts.models.gans package

```
class pl_bolts.models.gans.GAN(input_channels, input_height, input_width, latent_dim=32,
                                learning_rate=0.0002, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Vanilla GAN implementation.

Example:

```

from pl_bolts.models.gan import GAN

m = GAN()
Trainer(gpus=2).fit(m)

```

Example CLI:

```

# mnist
python basic_gan_module.py --gpus 1

# imagenet
python basic_gan_module.py --gpus 1 --dataset 'imagenet2012'
--data_dir /path/to/imagenet/folder/ --meta_dir ~/path/to/meta/bin/folder
--batch_size 256 --learning_rate 0.0001

```

Parameters

- **input_channels** (`int`) – number of channels of an image
- **input_height** (`int`) – image height
- **input_width** (`int`) – image width
- **latent_dim** (`int`) – emb dim for encoder
- **learning_rate** (`float`) – the learning rate

```
static add_model_specific_args(parent_parser)
```

```
configure_optimizers()
```

```
discriminator_loss(x)
```

```
discriminator_step(x)
```

forward (*z*)

Generates an image given input noise *z*

Example:

```
z = torch.rand(batch_size, latent_dim)
gan = GAN.load_from_checkpoint(PATH)
img = gan(z)
```

generator_loss (*x*)

generator_step (*x*)

init_discriminator (*img_dim*)

init_generator (*img_dim*)

training_step (*batch, batch_idx, optimizer_idx*)

```
class pl_bolts.models.gans.DCGAN (beta1=0.5, feature_maps_gen=64, feature_maps_disc=64,
                                  image_channels=1, latent_dim=100, learning_rate=0.0002,
                                  **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

DCGAN implementation.

Example:

```
from pl_bolts.models.gan import DCGAN

m = DCGAN()
Trainer(gpus=2).fit(m)
```

Example CLI:

```
# mnist
python dcgan_module.py --gpus 1

# cifar10
python dcgan_module.py --gpus 1 --dataset cifar10 --image_channels 3
```

Parameters

- **beta1** (*float*) – Beta1 value for Adam optimizer
- **feature_maps_gen** (*int*) – Number of feature maps to use for the generator
- **feature_maps_disc** (*int*) – Number of feature maps to use for the discriminator
- **image_channels** (*int*) – Number of channels of the images from the dataset
- **latent_dim** (*int*) – Dimension of the latent space
- **learning_rate** (*float*) – Learning rate

_disc_step (*real*)

Return type `Tensor`

_gen_step (*real*)

Return type `Tensor`

_get_disc_loss (*real*)

Return type `Tensor`

`_get_discriminator()`

Return type `Module`

`_get_fake_pred(real)`

Return type `Tensor`

`_get_gen_loss(real)`

Return type `Tensor`

`_get_generator()`

Return type `Module`

`_get_noise(n_samples, latent_dim)`

Return type `Tensor`

`static _weights_init(m)`

`static add_model_specific_args(parent_parser)`

Return type `ArgumentParser`

`configure_optimizers()`

`forward(noise)`
Generates an image given input noise

Example:

```
noise = torch.rand(batch_size, latent_dim)
gan = GAN.load_from_checkpoint(PATH)
img = gan(noise)
```

Return type `Tensor`

`training_step(batch, batch_idx, optimizer_idx)`

Subpackages

`pl_bolts.models.gans.basic` package

Submodules

`pl_bolts.models.gans.basic.basic_gan_module` module

`class pl_bolts.models.gans.basic.basic_gan_module.GAN(input_channels, input_height, input_width, latent_dim=32, learning_rate=0.0002, **kwargs)`

Bases: `pytorch_lightning.LightningModule`

Vanilla GAN implementation.

Example:

```
from pl_bolts.models.gan import GAN

m = GAN()
Trainer(gpus=2).fit(m)
```

Example CLI:

```
# mnist
python basic_gan_module.py --gpus 1

# imagenet
python basic_gan_module.py --gpus 1 --dataset 'imagenet2012'
--data_dir /path/to/imagenet/folder/ --meta_dir ~/path/to/meta/bin/folder
--batch_size 256 --learning_rate 0.0001
```

Parameters

- **input_channels** (*int*) – number of channels of an image
- **input_height** (*int*) – image height
- **input_width** (*int*) – image width
- **latent_dim** (*int*) – emb dim for encoder
- **learning_rate** (*float*) – the learning rate

static add_model_specific_args (*parent_parser*)

configure_optimizers ()

discriminator_loss (*x*)

discriminator_step (*x*)

forward (*z*)

Generates an image given input noise *z*

Example:

```
z = torch.rand(batch_size, latent_dim)
gan = GAN.load_from_checkpoint(PATH)
img = gan(z)
```

generator_loss (*x*)

generator_step (*x*)

init_discriminator (*img_dim*)

init_generator (*img_dim*)

training_step (*batch, batch_idx, optimizer_idx*)

`pl_bolts.models.gans.basic.basic_gan_module.cli_main` (*args=None*)

pl_bolts.models.gans.basic.components module

```
class pl_bolts.models.gans.basic.components.Discriminator (img_shape,          hid-
                                                    den_dim=1024)
    Bases: torch.nn.Module
    forward (img)

class pl_bolts.models.gans.basic.components.Generator (latent_dim, img_shape, hid-
                                                    den_dim=256)
    Bases: torch.nn.Module
    forward (z)
```

pl_bolts.models.gans.dcgan package**Submodules****pl_bolts.models.gans.dcgan.components module**

```
class pl_bolts.models.gans.dcgan.components.DCGANDiscriminator (feature_maps,
                                                                    im-
                                                                    age_channels)
    Bases: torch.nn.Module

    Parameters
    • feature_maps (int) – Number of feature maps to use
    • image_channels (int) – Number of channels of the images from the dataset

    static _make_disc_block (in_channels, out_channels, kernel_size=4, stride=2, padding=1,
                              bias=False, batch_norm=True, last_block=False)

    Return type Sequential

    forward (x)

    Return type Tensor

class pl_bolts.models.gans.dcgan.components.DCGANGenerator (latent_dim,          fea-
                                                                    ture_maps,          im-
                                                                    age_channels)
    Bases: torch.nn.Module

    Parameters
    • latent_dim (int) – Dimension of the latent space
    • feature_maps (int) – Number of feature maps to use
    • image_channels (int) – Number of channels of the images from the dataset

    static _make_gen_block (in_channels, out_channels, kernel_size=4, stride=2, padding=1,
                              bias=False, last_block=False)

    Return type Sequential

    forward (noise)

    Return type Tensor
```

pl_bolts.models.gans.dcgan.dcgan_module module

```
class pl_bolts.models.gans.dcgan.dcgan_module.DCGAN(beta1=0.5, feature_maps_gen=64, feature_maps_disc=64, image_channels=1, latent_dim=100, learning_rate=0.0002, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

DCGAN implementation.

Example:

```
from pl_bolts.models.gan import DCGAN

m = DCGAN()
Trainer(gpus=2).fit(m)
```

Example CLI:

```
# mnist
python dcgan_module.py --gpus 1

# cifar10
python dcgan_module.py --gpus 1 --dataset cifar10 --image_channels 3
```

Parameters

- **beta1** (`float`) – Beta1 value for Adam optimizer
- **feature_maps_gen** (`int`) – Number of feature maps to use for the generator
- **feature_maps_disc** (`int`) – Number of feature maps to use for the discriminator
- **image_channels** (`int`) – Number of channels of the images from the dataset
- **latent_dim** (`int`) – Dimension of the latent space
- **learning_rate** (`float`) – Learning rate

_disc_step (`real`)

Return type `Tensor`

_gen_step (`real`)

Return type `Tensor`

_get_disc_loss (`real`)

Return type `Tensor`

_get_discriminator ()

Return type `Module`

_get_fake_pred (`real`)

Return type `Tensor`

_get_gen_loss (`real`)

Return type `Tensor`

```
_get_generator()
```

Return type `Module`

```
_get_noise(n_samples, latent_dim)
```

Return type `Tensor`

```
static _weights_init(m)
```

```
static add_model_specific_args(parent_parser)
```

Return type `ArgumentParser`

```
configure_optimizers()
```

```
forward(noise)
```

Generates an image given input noise

Example:

```
noise = torch.rand(batch_size, latent_dim)
gan = GAN.load_from_checkpoint(PATH)
img = gan(noise)
```

Return type `Tensor`

```
training_step(batch, batch_idx, optimizer_idx)
```

```
pl_bolts.models.gans.dcgan.dcgan_module.cli_main(args=None)
```

pl_bolts.models.regression package

```
class pl_bolts.models.regression.LinearRegression(input_dim, output_dim=1,
                                                    bias=True, learning_rate=0.0001,
                                                    optimizer=torch.optim.Adam,
                                                    l1_strength=0.0, l2_strength=0.0,
                                                    **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Linear regression model implementing - with optional L1/L2 regularization $\min_{\{W\}} \|(Wx + b) - y\|_2^2$

Parameters

- **input_dim** (`int`) – number of dimensions of the input (1+)
- **output_dim** (`int`) – number of dimensions of the output (default=1)
- **bias** (`bool`) – If false, will not use $+b$
- **learning_rate** (`float`) – learning_rate for the optimizer
- **optimizer** (`Optimizer`) – the optimizer to use (default='Adam')
- **l1_strength** (`float`) – L1 regularization strength (default=None)
- **l2_strength** (`float`) – L2 regularization strength (default=None)

```
static add_model_specific_args(parent_parser)
```

```
configure_optimizers()
```

```
forward(x)
```

```
test_epoch_end(outputs)
test_step(batch, batch_idx)
training_step(batch, batch_idx)
validation_epoch_end(outputs)
validation_step(batch, batch_idx)

class pl_bolts.models.regression.LogisticRegression(input_dim, num_classes,
                                                    bias=True, learning_rate=0.0001,
                                                    optimizer=torch.optim.Adam,
                                                    l1_strength=0.0,
                                                    l2_strength=0.0, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Logistic regression model

Parameters

- **input_dim** (`int`) – number of dimensions of the input (at least 1)
- **num_classes** (`int`) – number of class labels (binary: 2, multi-class: >2)
- **bias** (`bool`) – specifies if a constant or intercept should be fitted (equivalent to `fit_intercept` in `sklearn`)
- **learning_rate** (`float`) – learning_rate for the optimizer
- **optimizer** (`Optimizer`) – the optimizer to use (default='Adam')
- **l1_strength** (`float`) – L1 regularization strength (default=None)
- **l2_strength** (`float`) – L2 regularization strength (default=None)

```
static add_model_specific_args(parent_parser)
```

```
configure_optimizers()
```

```
forward(x)
```

```
test_epoch_end(outputs)
```

```
test_step(batch, batch_idx)
```

```
training_step(batch, batch_idx)
```

```
validation_epoch_end(outputs)
```

```
validation_step(batch, batch_idx)
```

Submodules

pl_bolts.models.regression.linear_regression module

```

class pl_bolts.models.regression.linear_regression.LinearRegression(input_dim,
                                                                    out-
                                                                    put_dim=1,
                                                                    bias=True,
                                                                    learn-
                                                                    ing_rate=0.0001,
                                                                    opti-
                                                                    mizer=torch.optim.Adam,
                                                                    l1_strength=0.0,
                                                                    l2_strength=0.0,
                                                                    **kwargs)

```

Bases: `pytorch_lightning.LightningModule`

Linear regression model implementing - with optional L1/L2 regularization $\min_{\{W\}} \|(Wx + b) - y\|_2^2$

Parameters

- **input_dim** (`int`) – number of dimensions of the input (1+)
- **output_dim** (`int`) – number of dimensions of the output (default=1)
- **bias** (`bool`) – If false, will not use $+b$
- **learning_rate** (`float`) – learning_rate for the optimizer
- **optimizer** (`Optimizer`) – the optimizer to use (default='Adam')
- **l1_strength** (`float`) – L1 regularization strength (default=None)
- **l2_strength** (`float`) – L2 regularization strength (default=None)

static add_model_specific_args (`parent_parser`)

configure_optimizers ()

forward (`x`)

test_epoch_end (`outputs`)

test_step (`batch`, `batch_idx`)

training_step (`batch`, `batch_idx`)

validation_epoch_end (`outputs`)

validation_step (`batch`, `batch_idx`)

`pl_bolts.models.regression.linear_regression.cli_main()`

pl_bolts.models.regression.logistic_regression module

```
class pl_bolts.models.regression.logistic_regression.LogisticRegression(input_dim,
                                                                           num_classes,
                                                                           bias=True,
                                                                           learning_rate=0.0001,
                                                                           optimizer=torch.optim.Adam,
                                                                           l1_strength=0.0,
                                                                           l2_strength=0.0,
                                                                           **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Logistic regression model

Parameters

- **input_dim** (`int`) – number of dimensions of the input (at least 1)
- **num_classes** (`int`) – number of class labels (binary: 2, multi-class: >2)
- **bias** (`bool`) – specifies if a constant or intercept should be fitted (equivalent to `fit_intercept` in `sklearn`)
- **learning_rate** (`float`) – learning_rate for the optimizer
- **optimizer** (`Optimizer`) – the optimizer to use (default='Adam')
- **l1_strength** (`float`) – L1 regularization strength (default=None)
- **l2_strength** (`float`) – L2 regularization strength (default=None)

static add_model_specific_args (`parent_parser`)

configure_optimizers ()

forward (`x`)

test_epoch_end (`outputs`)

test_step (`batch`, `batch_idx`)

training_step (`batch`, `batch_idx`)

validation_epoch_end (`outputs`)

validation_step (`batch`, `batch_idx`)

`pl_bolts.models.regression.logistic_regression.cli_main()`

pl_bolts.models.rl package

```
class pl_bolts.models.rl.DoubleDQN(env,
                                     eps_start=1.0,
                                     eps_end=0.02,
                                     eps_last_frame=150000,
                                     sync_rate=1000,
                                     gamma=0.99,
                                     learning_rate=0.0001,
                                     batch_size=32,
                                     replay_size=100000,
                                     warm_start_size=10000,
                                     avg_reward_len=100,
                                     min_episode_reward=-21,
                                     seed=123,
                                     batches_per_epoch=1000,
                                     n_steps=1,
                                     **kwargs)
```

Bases: `pl_bolts.models.rl.dqn_model.DQN`

Double Deep Q-network (DDQN) PyTorch Lightning implementation of [Double DQN](#)

Paper authors: Hado van Hasselt, Arthur Guez, David Silver

Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.double_dqn_model import DoubleDQN
...
>>> model = DoubleDQN("PongNoFrameskip-v4")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: This example is based on https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter08/03_dqn_double.py

Note: Currently only supports CPU and single GPU training with *distributed_backend=dp*

Parameters

- **env** (*str*) – gym environment tag
- **eps_start** (*float*) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (*float*) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (*int*) – the final frame in for the decrease of epsilon. At this frame `epsilon = eps_end`
- **sync_rate** (*int*) – the number of iterations between syncing up the target network with the train network
- **gamma** (*float*) – discount factor
- **learning_rate** (*float*) – learning rate
- **batch_size** (*int*) – size of minibatch pulled from the DataLoader
- **replay_size** (*int*) – total capacity of the replay buffer
- **warm_start_size** (*int*) – how many random steps through the environment to be carried out at the start of training to fill the buffer with a starting point
- **avg_reward_len** (*int*) – how many episodes to take into account when calculating the avg reward
- **min_episode_reward** (*int*) – the minimum score that can be achieved in an episode. Used for filling the avg buffer before training begins
- **seed** (*int*) – seed value for all RNG used
- **batches_per_epoch** (*int*) – number of batches per epoch

- **n_steps** (`int`) – size of n step look ahead

training_step (`batch`, `_`)

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch recieved

Parameters

- **batch** (`Tuple[Tensor, Tensor]`) – current mini batch of replay data
- **_** – batch number, not used

Return type `OrderedDict`

Returns Training loss and log metrics

```
class pl_bolts.models.rl.DQN(env, eps_start=1.0, eps_end=0.02, eps_last_frame=150000,  
                             sync_rate=1000, gamma=0.99, learning_rate=0.0001,  
                             batch_size=32, replay_size=100000, warm_start_size=10000,  
                             avg_reward_len=100, min_episode_reward=-21, seed=123,  
                             batches_per_epoch=1000, n_steps=1, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Basic DQN Model

PyTorch Lightning implementation of [DQN](#) Paper authors: Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.dqn_model import DQN  
...  
>>> model = DQN("PongNoFrameskip-v4")
```

Train:

```
trainer = Trainer()  
trainer.fit(model)
```

Note: This example is based on: https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter06/02_dqn_pong.py

Note: Currently only supports CPU and single GPU training with `distributed_backend=dp`

Parameters

- **env** (`str`) – gym environment tag
- **eps_start** (`float`) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (`float`) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (`int`) – the final frame in for the decrease of epsilon. At this frame `epsilon = eps_end`

- **sync_rate** (*int*) – the number of iterations between syncing up the target network with the train network
- **gamma** (*float*) – discount factor
- **learning_rate** (*float*) – learning rate
- **batch_size** (*int*) – size of minibatch pulled from the DataLoader
- **replay_size** (*int*) – total capacity of the replay buffer
- **warm_start_size** (*int*) – how many random steps through the environment to be carried out at the start of training to fill the buffer with a starting point
- **avg_reward_len** (*int*) – how many episodes to take into account when calculating the avg reward
- **min_episode_reward** (*int*) – the minimum score that can be achieved in an episode. Used for filling the avg buffer before training begins
- **seed** (*int*) – seed value for all RNG used
- **batches_per_epoch** (*int*) – number of batches per epoch
- **n_steps** (*int*) – size of n step look ahead

_dataloader ()

Initialize the Replay Buffer dataset used for retrieving experiences

Return type `DataLoader`

static add_model_specific_args (*arg_parser*)

Adds arguments for DQN model

Note: These params are fine tuned for Pong env.

Parameters **arg_parser** (`ArgumentParser`) – parent parser

Return type `ArgumentParser`

build_networks ()

Initializes the DQN train and target networks

Return type `None`

configure_optimizers ()

Initialize Adam optimizer

Return type `List[Optimizer]`

forward (*x*)

Passes in a state *x* through the network and gets the q_values of each action as an output

Parameters **x** (`Tensor`) – environment state

Return type `Tensor`

Returns q values

static make_environment (*env_name*, *seed=None*)

Initialise gym environment

Parameters

- **env_name** (`str`) – environment name or tag
- **seed** (`Optional[int]`) – value to seed the environment RNG for reproducibility

Return type `object`

Returns gym environment

populate (*warm_start*)

Populates the buffer with initial experience

Return type `None`

run_n_episodes (*env, n_episodes=1, epsilon=1.0*)

Carries out N episodes of the environment with the current agent

Parameters

- **env** – environment to use, either train environment or test environment
- **n_episodes** (`int`) – number of episodes to run
- **epsilon** (`float`) – epsilon value for DQN agent

Return type `List[int]`

test_dataloader ()

Get test loader

Return type `DataLoader`

test_epoch_end (*outputs*)

Log the avg of the test results

Return type `Dict[str, Tensor]`

test_step (**args, **kwargs*)

Evaluate the agent for 10 episodes

Return type `Dict[str, Tensor]`

train_batch ()

Contains the logic for generating a new batch of data to be passed to the DataLoader

Return type `Tuple[Tensor, Tensor, Tensor, Tensor, Tensor]`

Returns yields a Experience tuple containing the state, action, reward, done and next_state.

train_dataloader ()

Get train loader

Return type `DataLoader`

training_step (*batch, _*)

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch recieved

Parameters

- **batch** (`Tuple[Tensor, Tensor]`) – current mini batch of replay data
- **_** – batch number, not used

Return type `OrderedDict`

Returns Training loss and log metrics

```
class pl_bolts.models.rl.DuelingDQN(env,
                                   eps_start=1.0,      eps_end=0.02,
                                   eps_last_frame=150000, sync_rate=1000,
                                   gamma=0.99, learning_rate=0.0001, batch_size=32,
                                   replay_size=100000, warm_start_size=10000,
                                   avg_reward_len=100, min_episode_reward=-21,
                                   seed=123, batches_per_epoch=1000, n_steps=1,
                                   **kwargs)
```

Bases: `pl_bolts.models.rl.dqn_model.DQN`

PyTorch Lightning implementation of [Dueling DQN](#)

Paper authors: Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas

Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.dueling_dqn_model import DuelingDQN
...
>>> model = DuelingDQN("PongNoFrameskip-v4")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: Currently only supports CPU and single GPU training with `distributed_backend=dp`

Parameters

- **env** (`str`) – gym environment tag
- **eps_start** (`float`) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (`float`) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (`int`) – the final frame in for the decrease of epsilon. At this frame `epsilon = eps_end`
- **sync_rate** (`int`) – the number of iterations between syncing up the target network with the train network
- **gamma** (`float`) – discount factor
- **learning_rate** (`float`) – learning rate
- **batch_size** (`int`) – size of minibatch pulled from the DataLoader
- **replay_size** (`int`) – total capacity of the replay buffer
- **warm_start_size** (`int`) – how many random steps through the environment to be carried out at the start of training to fill the buffer with a starting point
- **avg_reward_len** (`int`) – how many episodes to take into account when calculating the avg reward
- **min_episode_reward** (`int`) – the minimum score that can be achieved in an episode. Used for filling the avg buffer before training begins

- **seed** (*int*) – seed value for all RNG used
- **batches_per_epoch** (*int*) – number of batches per epoch
- **n_steps** (*int*) – size of n step look ahead

build_networks()

Initializes the Dueling DQN train and target networks

Return type None

```
class pl_bolts.models.rl.NoisyDQN(env, eps_start=1.0, eps_end=0.02, eps_last_frame=150000,  
                                sync_rate=1000,      gamma=0.99,      learning_rate=0.0001,  
                                batch_size=32,      replay_size=100000,  
                                warm_start_size=10000,  
                                avg_reward_len=100,  min_episode_reward=-21,  
                                seed=123,    batches_per_epoch=1000,  n_steps=1,  
                                **kwargs)
```

Bases: `pl_bolts.models.rl.dqn_model.DQN`

PyTorch Lightning implementation of **Noisy DQN**

Paper authors: Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, Shane Legg

Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.noisy_dqn_model import NoisyDQN  
...  
>>> model = NoisyDQN("PongNoFrameskip-v4")
```

Train:

```
trainer = Trainer()  
trainer.fit(model)
```

Note: Currently only supports CPU and single GPU training with `distributed_backend=dp`

Parameters

- **env** (*str*) – gym environment tag
- **eps_start** (*float*) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (*float*) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (*int*) – the final frame in for the decrease of epsilon. At this frame `epsilon = eps_end`
- **sync_rate** (*int*) – the number of iterations between syncing up the target network with the train network
- **gamma** (*float*) – discount factor
- **learning_rate** (*float*) – learning rate

- **batch_size** (*int*) – size of minibatch pulled from the DataLoader
- **replay_size** (*int*) – total capacity of the replay buffer
- **warm_start_size** (*int*) – how many random steps through the environment to be carried out at the start of training to fill the buffer with a starting point
- **avg_reward_len** (*int*) – how many episodes to take into account when calculating the avg reward
- **min_episode_reward** (*int*) – the minimum score that can be achieved in an episode. Used for filling the avg buffer before training begins
- **seed** (*int*) – seed value for all RNG used
- **batches_per_epoch** (*int*) – number of batches per epoch
- **n_steps** (*int*) – size of n step look ahead

build_networks ()

Initializes the Noisy DQN train and target networks

Return type None

on_train_start ()

Set the agents epsilon to 0 as the exploration comes from the network

Return type None

train_batch ()

Contains the logic for generating a new batch of data to be passed to the DataLoader. This is the same function as the standard DQN except that we dont update epsilon as it is always 0. The exploration comes from the noisy network.

Return type `Tuple[Tensor, Tensor, Tensor, Tensor, Tensor]`

Returns yields a Experience tuple containing the state, action, reward, done and next_state.

```
class pl_bolts.models.rl.PERDQN(env, eps_start=1.0, eps_end=0.02, eps_last_frame=150000,
                               sync_rate=1000, gamma=0.99, learning_rate=0.0001,
                               batch_size=32, replay_size=100000, warm_start_size=10000,
                               avg_reward_len=100, min_episode_reward=-21, seed=123,
                               batches_per_epoch=1000, n_steps=1, **kwargs)
```

Bases: `pl_bolts.models.rl.dqn_model.DQN`

PyTorch Lightning implementation of [DQN With Prioritized Experience Replay](#)

Paper authors: Tom Schaul, John Quan, Ioannis Antonoglou, David Silver

Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.per_dqn_model import PERDQN
...
>>> model = PERDQN("PongNoFrameskip-v4")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note:

This example is based on: https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter08/05_dqn_prio_replay.py

Note: Currently only supports CPU and single GPU training with *distributed_backend=dp*

Parameters

- **env** (*str*) – gym environment tag
- **eps_start** (*float*) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (*float*) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (*int*) – the final frame in for the decrease of epsilon. At this frame $\epsilon = \epsilon_{\text{end}}$
- **sync_rate** (*int*) – the number of iterations between syncing up the target network with the train network
- **gamma** (*float*) – discount factor
- **learning_rate** (*float*) – learning rate
- **batch_size** (*int*) – size of minibatch pulled from the DataLoader
- **replay_size** (*int*) – total capacity of the replay buffer
- **warm_start_size** (*int*) – how many random steps through the environment to be carried out at the start of training to fill the buffer with a starting point
- **avg_reward_len** (*int*) – how many episodes to take into account when calculating the avg reward
- **min_episode_reward** (*int*) – the minimum score that can be achieved in an episode. Used for filling the avg buffer before training begins
- **seed** (*int*) – seed value for all RNG used
- **batches_per_epoch** (*int*) – number of batches per epoch
- **n_steps** (*int*) – size of n step look ahead

_dataloader()

Initialize the Replay Buffer dataset used for retrieving experiences

Return type `DataLoader`

train_batch()

Contains the logic for generating a new batch of data to be passed to the DataLoader

Return type `Tuple[Tensor, Tensor, Tensor, Tensor, Tensor]`

Returns yields a Experience tuple containing the state, action, reward, done and next_state.

training_step(batch, _)

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch recieved

Parameters

- **batch** – current mini batch of replay data
- **_** – batch number, not used

Return type `OrderedDict`

Returns Training loss and log metrics

```
class pl_bolts.models.rl.Reinforce(env, gamma=0.99, lr=0.01, batch_size=8,
                                   n_steps=10, avg_reward_len=100, entropy_beta=0.01,
                                   epoch_len=1000, num_batch_episodes=4, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of **REINFORCE** Paper authors: Richard S. Sutton, David McAllester, Satinder Singh, Yishay Mansour Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.reinforce_model import Reinforce
...
>>> model = Reinforce("CartPole-v0")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: This example is based on: https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter11/02_cartpole_reinforce.py

Note: Currently only supports CPU and single GPU training with *distributed_backend=dp*

Parameters

- **env** (`str`) – gym environment tag
- **gamma** (`float`) – discount factor
- **lr** (`float`) – learning rate
- **batch_size** (`int`) – size of minibatch pulled from the DataLoader
- **n_steps** (`int`) – number of stakes per discounted experience

- **entropy_beta** (*float*) – entropy coefficient
- **epoch_len** (*int*) – how many batches before pseudo epoch
- **num_batch_episodes** (*int*) – how many episodes to rollout for each batch of training
- **avg_reward_len** (*int*) – how many episodes to take into account when calculating the avg reward

_dataloader ()

Initialize the Replay Buffer dataset used for retrieving experiences

Return type `DataLoader`

static add_model_specific_args (*arg_parser*)

Adds arguments for DQN model

Note: These params are fine tuned for Pong env.

Parameters **arg_parser** – the current argument parser to add to

Return type `ArgumentParser`

Returns **arg_parser** with model specific args added

calc_qvals (*rewards*)

Calculate the discounted rewards of all rewards in list

Parameters **rewards** (`List[float]`) – list of rewards from latest batch

Return type `List[float]`

Returns list of discounted rewards

configure_optimizers ()

Initialize Adam optimizer

Return type `List[Optimizer]`

discount_rewards (*experiences*)

Calculates the discounted reward over N experiences

Parameters **experiences** (`Tuple[Experience]`) – Tuple of Experience

Return type `float`

Returns total discounted reward

forward (*x*)

Passes in a state x through the network and gets the q_values of each action as an output

Parameters **x** (`Tensor`) – environment state

Return type `Tensor`

Returns q values

get_device (*batch*)

Retrieve device currently being used by minibatch

Return type `str`

loss (*states, actions, scaled_rewards*)

Return type `Tensor`

train_batch()

Contains the logic for generating a new batch of data to be passed to the DataLoader

Yields yields a tuple of Lists containing tensors for states, actions and rewards of the batch.

Return type `Tuple[List[Tensor], List[Tensor], List[Tensor]]`

train_dataloader()

Get train loader

Return type `DataLoader`

training_step(batch, _)

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch recieved

Parameters

- **batch** (`Tuple[Tensor, Tensor]`) – current mini batch of replay data
- **_** – batch number, not used

Return type `OrderedDict`

Returns Training loss and log metrics

```
class pl_bolts.models.rl.VanillaPolicyGradient(env, gamma=0.99, lr=0.01,
                                              batch_size=8, n_steps=10,
                                              avg_reward_len=100, entropy_beta=0.01, epoch_len=1000,
                                              **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of `Vanilla Policy Gradient`

Paper authors: Richard S. Sutton, David McAllester, Satinder Singh, Yishay Mansour

Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.vanilla_policy_gradient_model import _
    ↪ VanillaPolicyGradient
    ...
>>> model = VanillaPolicyGradient("CartPole-v0")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: This example is based on: https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter11/04_cartpole_pg.py

Note: Currently only supports CPU and single GPU training with *distributed_backend=dp*

Parameters

- **env** (*str*) – gym environment tag
- **gamma** (*float*) – discount factor
- **lr** (*float*) – learning rate
- **batch_size** (*int*) – size of minibatch pulled from the DataLoader
- **batch_episodes** – how many episodes to rollout for each batch of training
- **entropy_beta** (*float*) – dictates the level of entropy per batch
- **avg_reward_len** (*int*) – how many episodes to take into account when calculating the avg reward
- **epoch_len** (*int*) – how many batches before pseudo epoch

_dataloader ()

Initialize the Replay Buffer dataset used for retrieving experiences

Return type *DataLoader*

static add_model_specific_args (*arg_parser*)

Adds arguments for DQN model

Note: These params are fine tuned for Pong env.

Parameters **arg_parser** – the current argument parser to add to

Return type *ArgumentParser*

Returns **arg_parser** with model specific args added

compute_returns (*rewards*)

Calculate the discounted rewards of the batched rewards

Parameters **rewards** – list of batched rewards

Returns list of discounted rewards

configure_optimizers ()

Initialize Adam optimizer

Return type *List[Optimizer]*

forward (*x*)

Passes in a state *x* through the network and gets the q_values of each action as an output

Parameters **x** (*Tensor*) – environment state

Return type *Tensor*

Returns q values

get_device (*batch*)

Retrieve device currently being used by minibatch

Return type `str`

loss (*states, actions, scaled_rewards*)

Calculates the loss for VPG

Parameters

- **states** – batched states
- **actions** – batch actions
- **scaled_rewards** – batched Q values

Return type `Tensor`

Returns loss for the current batch

train_batch ()

Contains the logic for generating a new batch of data to be passed to the DataLoader

Return type `Tuple[List[Tensor], List[Tensor], List[Tensor]]`

Returns yields a tuple of Lists containing tensors for states, actions and rewards of the batch.

train_dataloader ()

Get train loader

Return type `DataLoader`

training_step (*batch, _*)

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch received

Parameters

- **batch** (`Tuple[Tensor, Tensor]`) – current mini batch of replay data
- **_** – batch number, not used

Return type `OrderedDict`

Returns Training loss and log metrics

Subpackages

`pl_bolts.models.rl.common` package

Submodules

`pl_bolts.models.rl.common.agents` module

Agent module containing classes for Agent logic Based on the implementations found here: <https://github.com/Shmuma/ptan/blob/master/ptan/agent.py>

class `pl_bolts.models.rl.common.agents.Agent` (*net*)

Bases: `abc.ABC`

Basic agent that always returns 0

__call__ (*state, device, *args, **kwargs*)

Using the given network, decide what action to carry

Parameters

- **state** (`Tensor`) – current state of the environment
- **device** (`str`) – device used for current batch

Return type `List[int]`

Returns action

class `pl_bolts.models.rl.common.agents.PolicyAgent` (*net*)
Bases: `pl_bolts.models.rl.common.agents.Agent`

Policy based agent that returns an action based on the networks policy

__call__ (*states, device*)

Takes in the current state and returns the action based on the agents policy

Parameters

- **states** (`Tensor`) – current state of the environment
- **device** (`str`) – the device used for the current batch

Return type `List[int]`

Returns action defined by policy

class `pl_bolts.models.rl.common.agents.ValueAgent` (*net, action_space, eps_start=1.0, eps_end=0.2, eps_frames=1000*)
Bases: `pl_bolts.models.rl.common.agents.Agent`

Value based agent that returns an action based on the Q values from the network

__call__ (*state, device*)

Takes in the current state and returns the action based on the agents policy

Parameters

- **state** (`Tensor`) – current state of the environment
- **device** (`str`) – the device used for the current batch

Return type `List[int]`

Returns action defined by policy

get_action (*state, device*)

Returns the best action based on the Q values of the network

Parameters

- **state** (`Tensor`) – current state of the environment
- **device** (`device`) – the device used for the current batch

Returns action defined by Q values

get_random_action (*state*)
returns a random action

Return type `int`

update_epsilon (*step*)

Updates the epsilon value based on the current step

Parameters **step** (`int`) – current global step

Return type `None`

pl_bolts.models.rl.common.cli module

Contains generic arguments used for all models

`pl_bolts.models.rl.common.cli.add_base_args` (*parent*)
Adds arguments for DQN model

Note: These params are fine tuned for Pong env.

Parameters *parent* –

Return type `ArgumentParser`

pl_bolts.models.rl.common.gym_wrappers module

Set of wrapper functions for gym environments taken from <https://github.com/Shmuma/ptan/blob/master/ptan/common/wrappers.py>

class `pl_bolts.models.rl.common.gym_wrappers.BufferWrapper` (*env*, *n_steps*,
dtype=<class
'numpy.float32'>)

Bases: `object`

“Wrapper for image stacking

observation (*observation*)
convert observation

reset ()
reset env

class `pl_bolts.models.rl.common.gym_wrappers.DataAugmentation` (*env*=None)
Bases: `object`

Carries out basic data augmentation on the env observations - ToTensor - GrayScale - RandomCrop

observation (*obs*)
preprocess the obs

class `pl_bolts.models.rl.common.gym_wrappers.FireResetEnv` (*env*=None)
Bases: `object`

For environments where the user need to press FIRE for the game to start.

reset ()
reset the env

step (*action*)
Take 1 step

class `pl_bolts.models.rl.common.gym_wrappers.ImageToPyTorch` (*env*)
Bases: `object`

converts image to pytorch format

static observation (*observation*)
convert observation

```
class pl_bolts.models.rl.common.gym_wrappers.MaxAndSkipEnv (env=None, skip=4)
    Bases: object

    Return only every skip-th frame

    reset ()
        Clear past frame buffer and init. to first obs. from inner env.

    step (action)
        take 1 step

class pl_bolts.models.rl.common.gym_wrappers.ProcessFrame84 (env=None)
    Bases: object

    preprocessing images from env

    observation (obs)
        preprocess the obs

    static process (frame)
        image preprocessing, formats to 84x84

class pl_bolts.models.rl.common.gym_wrappers.ScaledFloatFrame
    Bases: object

    scales the pixels

    static observation (obs)

class pl_bolts.models.rl.common.gym_wrappers.ToTensor (env=None)
    Bases: object

    For environments where the user need to press FIRE for the game to start.

    reset ()
        reset the env and cast to tensor

    step (action)
        Take 1 step and cast to tensor

pl_bolts.models.rl.common.gym_wrappers.make_environment (env_name)
    Convert environment with wrappers
```

pl_bolts.models.rl.common.memory module

Series of memory buffers sued

```
class pl_bolts.models.rl.common.memory.Buffer (capacity)
    Bases: object

    Basic Buffer for storing a single experience at a time

    Parameters capacity (int) – size of the buffer

    append (experience)
        Add experience to the buffer

        Parameters experience (Experience) – tuple (state, action, reward, done, new_state)

        Return type None

    sample (*args)
        returns everything in the buffer so far it is then reset :rtype: Union[Tuple, List[Tuple]] :returns: a
        batch of tuple np arrays of state, action, reward, done, next_state
```

```

class pl_bolts.models.rl.common.memory.Experience(state, action, reward, done,
                                                    new_state)
    Bases: tuple
    Create new instance of Experience(state, action, reward, done, new_state)
    _asdict()
        Return a new OrderedDict which maps field names to their values.
    classmethod _make(iterable)
        Make a new Experience object from a sequence or iterable
    _replace(**kwargs)
        Return a new Experience object replacing specified fields with new values
    _field_defaults = {}
    _fields = ('state', 'action', 'reward', 'done', 'new_state')
    _fields_defaults = {}
    property action
        Alias for field number 1
    property done
        Alias for field number 3
    property new_state
        Alias for field number 4
    property reward
        Alias for field number 2
    property state
        Alias for field number 0

class pl_bolts.models.rl.common.memory.MeanBuffer(capacity)
    Bases: object
    Stores a deque of items and calculates the mean
    add(val)
        Add to the buffer
        Return type None
    mean()
        Retrieve the mean
        Return type float

class pl_bolts.models.rl.common.memory.MultiStepBuffer(capacity, n_steps=1,
                                                         gamma=0.99)
    Bases: pl_bolts.models.rl.common.memory.ReplayBuffer
    N Step Replay Buffer
    Parameters
        • capacity(int) – max number of experiences that will be stored in the buffer
        • n_steps(int) – number of steps used for calculating discounted reward/experience
        • gamma(float) – discount factor when calculating n_step discounted reward of the experience being stored in buffer

```

append (*exp*)

Add experience to the buffer

Parameters **exp** (*Experience*) – tuple (state, action, reward, done, new_state)

Return type None

discount_rewards (*experiences*)

Calculates the discounted reward over N experiences

Parameters **experiences** (*Tuple[Experience]*) – Tuple of Experience

Return type float

Returns total discounted reward

split_head_tail_exp (*experiences*)

Takes in a tuple of experiences and returns the last state and tail experiences based on if the last state is the end of an episode

Parameters **experiences** (*Tuple[Experience]*) – Tuple of N Experience

Return type *Tuple[List, Tuple[Experience]]*

Returns last state (Array or None) and remaining Experience

update_history_queue (*exp*)

Updates the experience history queue with the latest experiences. In the event of an experience step is in the done state, the history will be incrementally appended to the queue, removing the tail of the history each time.

Parameters

- **env_idx** – index of the environment
- **exp** – the current experience
- **history** – history of experience steps for this environment

Return type None

class `pl_bolts.models.rl.common.memory.PERBuffer` (*buffer_size*, *prob_alpha=0.6*,
beta_start=0.4,
beta_frames=100000)

Bases: `pl_bolts.models.rl.common.memory.ReplayBuffer`

simple list based Prioritized Experience Replay Buffer Based on implementation found here: <https://github.com/Shmuma/ptan/blob/master/ptan/experience.py#L371>

Args: capacity: size of the buffer

append (*exp*)

Adds experiences from exp_source to the PER buffer

Parameters **exp** – experience tuple being added to the buffer

Return type None

sample (*batch_size=32*)

Takes a prioritized sample from the buffer

Parameters **batch_size** – size of sample

Return type *Tuple*

Returns sample of experiences chosen with ranked probability

update_beta (*step*)

Update the beta value which accounts for the bias in the PER

Parameters **step** – current global step

Return type `float`

Returns beta value for this indexed experience

update_priorities (*batch_indices, batch_priorities*)

Update the priorities from the last batch, this should be called after the loss for this batch has been calculated.

Parameters

- **batch_indices** (`List`) – index of each datum in the batch
- **batch_priorities** (`List`) – priority of each datum in the batch

Return type `None`

class `pl_bolts.models.rl.common.memory.ReplayBuffer` (*capacity*)

Bases: `pl_bolts.models.rl.common.memory.Buffer`

Replay Buffer for storing past experiences allowing the agent to learn from them

Parameters **capacity** (`int`) – size of the buffer

sample (*batch_size*)

Takes a sample of the buffer

Parameters **batch_size** (`int`) – current batch_size

Return type `Tuple`

Returns a batch of tuple np arrays of state, action, reward, done, next_state

`pl_bolts.models.rl.common.networks` module

Series of networks used Based on implementations found here:

class `pl_bolts.models.rl.common.networks.CNN` (*input_shape, n_actions*)

Bases: `torch.nn.Module`

Simple MLP network

Parameters

- **input_shape** (`Tuple[int]`) – observation shape of the environment
- **n_actions** (`int`) – number of discrete actions available in the environment

_get_conv_out (*shape*)

Calculates the output size of the last conv layer

Parameters **shape** – input dimensions

Return type `int`

Returns size of the conv output

forward (*input_x*)

Forward pass through network

Parameters **x** – input to network

Return type `Tensor`

Returns output of network

class `pl_bolts.models.rl.common.networks.DuelingCNN` (*input_shape*, *n_actions*, *_=128*)

Bases: `torch.nn.Module`

CNN network with duel heads for val and advantage

Parameters

- **input_shape** (`Tuple[int]`) – observation shape of the environment
- **n_actions** (`int`) – number of discrete actions available in the environment
- **hidden_size** – size of hidden layers

_get_conv_out (*shape*)

Calculates the output size of the last conv layer

Parameters **shape** – input dimensions

Return type `int`

Returns size of the conv output

adv_val (*input_x*)

Gets the advantage and value by passing out of the base network through the value and advantage heads

Parameters **input_x** – input to network

Returns advantage, value

forward (*input_x*)

Forward pass through network. Calculates the Q using the value and advantage

Parameters **input_x** – input to network

Returns Q value

class `pl_bolts.models.rl.common.networks.DuelingMLP` (*input_shape*, *n_actions*, *hidden_size=128*)

Bases: `torch.nn.Module`

MLP network with duel heads for val and advantage

Parameters

- **input_shape** (`Tuple[int]`) – observation shape of the environment
- **n_actions** (`int`) – number of discrete actions available in the environment
- **hidden_size** (`int`) – size of hidden layers

adv_val (*input_x*)

Gets the advantage and value by passing out of the base network through the value and advantage heads

Parameters **input_x** – input to network

Return type `Tuple[Tensor, Tensor]`

Returns advantage, value

forward (*input_x*)

Forward pass through network. Calculates the Q using the value and advantage

Parameters **x** – input to network

Returns Q value

class `pl_bolts.models.rl.common.networks.MLP` (*input_shape, n_actions, hidden_size=128*)
 Bases: `torch.nn.Module`

Simple MLP network

Parameters

- **input_shape** (`Tuple[int]`) – observation shape of the environment
- **n_actions** (`int`) – number of discrete actions available in the environment
- **hidden_size** (`int`) – size of hidden layers

forward (*input_x*)

Forward pass through network

Parameters **x** – input to network

Returns output of network

class `pl_bolts.models.rl.common.networks.NoisyCNN` (*input_shape, n_actions*)
 Bases: `torch.nn.Module`

CNN with Noisy Linear layers for exploration

Parameters

- **input_shape** (`Tuple[int]`) – observation shape of the environment
- **n_actions** (`int`) – number of discrete actions available in the environment

_get_conv_out (*shape*)

Calculates the output size of the last conv layer

Parameters **shape** – input dimensions

Return type `int`

Returns size of the conv output

forward (*input_x*)

Forward pass through network

Parameters **x** – input to network

Return type `Tensor`

Returns output of network

class `pl_bolts.models.rl.common.networks.NoisyLinear` (*in_features, out_features, sigma_init=0.017, bias=True*)
 Bases: `torch.nn.Linear`

Noisy Layer using Independent Gaussian Noise. based on https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter08/lib/dqn_extra.py#L19

Parameters

- **in_features** (`int`) – number of inputs
- **out_features** (`int`) – number of outputs
- **sigma_init** (`float`) – initial fill value of noisy weights
- **bias** (`bool`) – flag to include bias to linear layer

forward (*input_x*)

Forward pass of the layer

Parameters `input_x` (`Tensor`) – input tensor

Return type `Tensor`

Returns output of the layer

reset_parameters ()

initializes or resets the parameter of the layer

Return type `None`

Submodules

`pl_bolts.models.rl.double_dqn_model` module

Double DQN

```
class pl_bolts.models.rl.double_dqn_model.DoubleDQN(env, eps_start=1.0,
                                                    eps_end=0.02,
                                                    eps_last_frame=150000,
                                                    sync_rate=1000, gamma=0.99,
                                                    learning_rate=0.0001,
                                                    batch_size=32, re-
                                                    play_size=100000,
                                                    warm_start_size=10000,
                                                    avg_reward_len=100,
                                                    min_episode_reward=-
                                                    21, seed=123,
                                                    batches_per_epoch=1000,
                                                    n_steps=1, **kwargs)
```

Bases: `pl_bolts.models.rl.dqn_model.DQN`

Double Deep Q-network (DDQN) PyTorch Lightning implementation of `Double DQN`

Paper authors: Hado van Hasselt, Arthur Guez, David Silver

Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.double_dqn_model import DoubleDQN
...
>>> model = DoubleDQN("PongNoFrameskip-v4")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: This example is based on https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter08/03_dqn_double.py

Note: Currently only supports CPU and single GPU training with *distributed_backend=dp*

Parameters

- **env** (`str`) – gym environment tag
- **eps_start** (`float`) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (`float`) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (`int`) – the final frame in for the decrease of epsilon. At this frame `epsilon = eps_end`
- **sync_rate** (`int`) – the number of iterations between syncing up the target network with the train network
- **gamma** (`float`) – discount factor
- **learning_rate** (`float`) – learning rate
- **batch_size** (`int`) – size of minibatch pulled from the DataLoader
- **replay_size** (`int`) – total capacity of the replay buffer
- **warm_start_size** (`int`) – how many random steps through the environment to be carried out at the start of training to fill the buffer with a starting point
- **avg_reward_len** (`int`) – how many episodes to take into account when calculating the avg reward
- **min_episode_reward** (`int`) – the minimum score that can be achieved in an episode. Used for filling the avg buffer before training begins
- **seed** (`int`) – seed value for all RNG used
- **batches_per_epoch** (`int`) – number of batches per epoch
- **n_steps** (`int`) – size of n step look ahead

`training_step(batch, _)`

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch recieved

Parameters

- **batch** (`Tuple[Tensor, Tensor]`) – current mini batch of replay data
- **_** – batch number, not used

Return type `OrderedDict`

Returns Training loss and log metrics

```
pl_bolts.models.rl.double_dqn_model.cli_main()
```

pl_bolts.models.rl.dqn_model module

Deep Q Network

```
class pl_bolts.models.rl.dqn_model.DQN(env,          eps_start=1.0,          eps_end=0.02,
                                         eps_last_frame=150000,          sync_rate=1000,
                                         gamma=0.99,          learning_rate=0.0001,
                                         batch_size=32,          replay_size=100000,
                                         warm_start_size=10000,          avg_reward_len=100,
                                         min_episode_reward=-21,          seed=123,
                                         batches_per_epoch=1000, n_steps=1, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Basic DQN Model

PyTorch Lightning implementation of [DQN](#) Paper authors: Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.dqn_model import DQN
...
>>> model = DQN("PongNoFrameskip-v4")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: This example is based on: https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter06/02_dqn_pong.py

Note: Currently only supports CPU and single GPU training with `distributed_backend=dp`

Parameters

- **env** (`str`) – gym environment tag
- **eps_start** (`float`) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (`float`) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (`int`) – the final frame in for the decrease of epsilon. At this frame `epsilon = eps_end`
- **sync_rate** (`int`) – the number of iterations between syncing up the target network with the train network
- **gamma** (`float`) – discount factor
- **learning_rate** (`float`) – learning rate
- **batch_size** (`int`) – size of minibatch pulled from the DataLoader

- **replay_size** (*int*) – total capacity of the replay buffer
- **warm_start_size** (*int*) – how many random steps through the environment to be carried out at the start of training to fill the buffer with a starting point
- **avg_reward_len** (*int*) – how many episodes to take into account when calculating the avg reward
- **min_episode_reward** (*int*) – the minimum score that can be achieved in an episode. Used for filling the avg buffer before training begins
- **seed** (*int*) – seed value for all RNG used
- **batches_per_epoch** (*int*) – number of batches per epoch
- **n_steps** (*int*) – size of n step look ahead

_dataloader ()

Initialize the Replay Buffer dataset used for retrieving experiences

Return type `DataLoader`

static add_model_specific_args (*arg_parser*)

Adds arguments for DQN model

Note: These params are fine tuned for Pong env.

Parameters **arg_parser** (`ArgumentParser`) – parent parser

Return type `ArgumentParser`

build_networks ()

Initializes the DQN train and target networks

Return type `None`

configure_optimizers ()

Initialize Adam optimizer

Return type `List[Optimizer]`

forward (*x*)

Passes in a state *x* through the network and gets the q_values of each action as an output

Parameters **x** (`Tensor`) – environment state

Return type `Tensor`

Returns q values

static make_environment (*env_name*, *seed=None*)

Initialise gym environment

Parameters

- **env_name** (*str*) – environment name or tag
- **seed** (`Optional[int]`) – value to seed the environment RNG for reproducibility

Return type `object`

Returns gym environment

populate (*warm_start*)

Populates the buffer with initial experience

Return type `None`

run_n_episodes (*env, n_episodes=1, epsilon=1.0*)

Carries out N episodes of the environment with the current agent

Parameters

- **env** – environment to use, either train environment or test environment
- **n_episodes** (`int`) – number of episodes to run
- **epsilon** (`float`) – epsilon value for DQN agent

Return type `List[int]`

test_dataloader ()

Get test loader

Return type `DataLoader`

test_epoch_end (*outputs*)

Log the avg of the test results

Return type `Dict[str, Tensor]`

test_step (**args, **kwargs*)

Evaluate the agent for 10 episodes

Return type `Dict[str, Tensor]`

train_batch ()

Contains the logic for generating a new batch of data to be passed to the DataLoader

Return type `Tuple[Tensor, Tensor, Tensor, Tensor, Tensor]`

Returns yields a Experience tuple containing the state, action, reward, done and next_state.

train_dataloader ()

Get train loader

Return type `DataLoader`

training_step (*batch, _*)

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch recieved

Parameters

- **batch** (`Tuple[Tensor, Tensor]`) – current mini batch of replay data
- **_** – batch number, not used

Return type `OrderedDict`

Returns Training loss and log metrics

`pl_bolts.models.rl.dqn_model.cli_main()`

pl_bolts.models.rl.dueling_dqn_model module

Dueling DQN

```
class pl_bolts.models.rl.dueling_dqn_model.DuelingDQN(env,
                                                    eps_start=1.0,
                                                    eps_end=0.02,
                                                    eps_last_frame=150000,
                                                    sync_rate=1000,
                                                    gamma=0.99,          learn-
                                                    ing_rate=0.0001,
                                                    batch_size=32,          re-
                                                    play_size=100000,
                                                    warm_start_size=10000,
                                                    avg_reward_len=100,
                                                    min_episode_reward=-
                                                    21,          seed=123,
                                                    batches_per_epoch=1000,
                                                    n_steps=1, **kwargs)
```

Bases: `pl_bolts.models.rl.dqn_model.DQN`

PyTorch Lightning implementation of [Dueling DQN](#)

Paper authors: Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas

Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.dueling_dqn_model import DuelingDQN
...
>>> model = DuelingDQN("PongNoFrameskip-v4")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: Currently only supports CPU and single GPU training with `distributed_backend=dp`

Parameters

- **env** (`str`) – gym environment tag
- **eps_start** (`float`) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (`float`) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (`int`) – the final frame in for the decrease of epsilon. At this frame `epsilon = eps_end`
- **sync_rate** (`int`) – the number of iterations between syncing up the target network with the train network
- **gamma** (`float`) – discount factor

- **learning_rate** (`float`) – learning rate
- **batch_size** (`int`) – size of minibatch pulled from the DataLoader
- **replay_size** (`int`) – total capacity of the replay buffer
- **warm_start_size** (`int`) – how many random steps through the environment to be carried out at the start of training to fill the buffer with a starting point
- **avg_reward_len** (`int`) – how many episodes to take into account when calculating the avg reward
- **min_episode_reward** (`int`) – the minimum score that can be achieved in an episode. Used for filling the avg buffer before training begins
- **seed** (`int`) – seed value for all RNG used
- **batches_per_epoch** (`int`) – number of batches per epoch
- **n_steps** (`int`) – size of n step look ahead

build_networks ()

Initializes the Dueling DQN train and target networks

Return type None

```
pl_bolts.models.rl.dueling_dqn_model.cli_main()
```

pl_bolts.models.rl.noisy_dqn_model module

Noisy DQN

```
class pl_bolts.models.rl.noisy_dqn_model.NoisyDQN(env, eps_start=1.0, eps_end=0.02,
                                                  eps_last_frame=150000,
                                                  sync_rate=1000, gamma=0.99,
                                                  learning_rate=0.0001,
                                                  batch_size=32,          re-
                                                  play_size=100000,
                                                  warm_start_size=10000,
                                                  avg_reward_len=100,
                                                  min_episode_reward=-
                                                  21,          seed=123,
                                                  batches_per_epoch=1000,
                                                  n_steps=1, **kwargs)
```

Bases: `pl_bolts.models.rl.dqn_model.DQN`

PyTorch Lightning implementation of **Noisy DQN**

Paper authors: Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, Shane Legg

Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.noisy_dqn_model import NoisyDQN
...
>>> model = NoisyDQN("PongNoFrameskip-v4")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: Currently only supports CPU and single GPU training with *distributed_backend=dp*

Parameters

- **env** (*str*) – gym environment tag
- **eps_start** (*float*) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (*float*) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (*int*) – the final frame in for the decrease of epsilon. At this frame $\epsilon = \epsilon_{\text{end}}$
- **sync_rate** (*int*) – the number of iterations between syncing up the target network with the train network
- **gamma** (*float*) – discount factor
- **learning_rate** (*float*) – learning rate
- **batch_size** (*int*) – size of minibatch pulled from the DataLoader
- **replay_size** (*int*) – total capacity of the replay buffer
- **warm_start_size** (*int*) – how many random steps through the environment to be carried out at the start of training to fill the buffer with a starting point
- **avg_reward_len** (*int*) – how many episodes to take into account when calculating the avg reward
- **min_episode_reward** (*int*) – the minimum score that can be achieved in an episode. Used for filling the avg buffer before training begins
- **seed** (*int*) – seed value for all RNG used
- **batches_per_epoch** (*int*) – number of batches per epoch
- **n_steps** (*int*) – size of n step look ahead

build_networks()

Initializes the Noisy DQN train and target networks

Return type None

on_train_start()

Set the agents epsilon to 0 as the exploration comes from the network

Return type None

train_batch()

Contains the logic for generating a new batch of data to be passed to the DataLoader. This is the same function as the standard DQN except that we don't update epsilon as it is always 0. The exploration comes from the noisy network.

Return type `Tuple[Tensor, Tensor, Tensor, Tensor, Tensor]`

Returns yields a Experience tuple containing the state, action, reward, done and next_state.

```
pl_bolts.models.rl.noisy_dqn_model.cli_main()
```

pl_bolts.models.rl.per_dqn_model module

Prioritized Experience Replay DQN

```
class pl_bolts.models.rl.per_dqn_model.PERDQN(env, eps_start=1.0, eps_end=0.02,
                                              eps_last_frame=150000,
                                              sync_rate=1000, gamma=0.99,
                                              learning_rate=0.0001,
                                              batch_size=32, replay_size=100000,
                                              warm_start_size=10000,
                                              avg_reward_len=100,
                                              min_episode_reward=-21, seed=123,
                                              batches_per_epoch=1000, n_steps=1,
                                              **kwargs)
```

Bases: `pl_bolts.models.rl.dqn_model.DQN`

PyTorch Lightning implementation of DQN With Prioritized Experience Replay

Paper authors: Tom Schaul, John Quan, Ioannis Antonoglou, David Silver

Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.per_dqn_model import PERDQN
...
>>> model = PERDQN("PongNoFrameskip-v4")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note:

This example is based on: https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter08/05_dqn_prio_replay.py

Note: Currently only supports CPU and single GPU training with `distributed_backend=dp`

Parameters

- **env** (*str*) – gym environment tag
- **eps_start** (*float*) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (*float*) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (*int*) – the final frame in for the decrease of epsilon. At this frame `epsilon = eps_end`
- **sync_rate** (*int*) – the number of iterations between syncing up the target network with the train network
- **gamma** (*float*) – discount factor
- **learning_rate** (*float*) – learning rate
- **batch_size** (*int*) – size of minibatch pulled from the DataLoader
- **replay_size** (*int*) – total capacity of the replay buffer
- **warm_start_size** (*int*) – how many random steps through the environment to be carried out at the start of training to fill the buffer with a starting point
- **avg_reward_len** (*int*) – how many episodes to take into account when calculating the avg reward
- **min_episode_reward** (*int*) – the minimum score that can be achieved in an episode. Used for filling the avg buffer before training begins
- **seed** (*int*) – seed value for all RNG used
- **batches_per_epoch** (*int*) – number of batches per epoch
- **n_steps** (*int*) – size of n step look ahead

_dataloader ()

Initialize the Replay Buffer dataset used for retrieving experiences

Return type `DataLoader`

train_batch ()

Contains the logic for generating a new batch of data to be passed to the DataLoader

Return type `Tuple[Tensor, Tensor, Tensor, Tensor, Tensor]`

Returns yields a Experience tuple containing the state, action, reward, done and next_state.

training_step (*batch*, *_*)

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch recieved

Parameters

- **batch** – current mini batch of replay data
- **_** – batch number, not used

Return type `OrderedDict`

Returns Training loss and log metrics

`pl_bolts.models.rl.per_dqn_model.cli_main()`

pl_bolts.models.rl.reinforce_model module

```
class pl_bolts.models.rl.reinforce_model.Reinforce(env, gamma=0.99, lr=0.01,
                                                    batch_size=8, n_steps=10,
                                                    avg_reward_len=100,
                                                    entropy_beta=0.01,
                                                    epoch_len=1000,
                                                    num_batch_episodes=4,
                                                    **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [REINFORCE](#) Paper authors: Richard S. Sutton, David McAllester, Satinder Singh, Yishay Mansour Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.reinforce_model import Reinforce
...
>>> model = Reinforce("CartPole-v0")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: This example is based on: https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter11/02_cartpole_reinforce.py

Note: Currently only supports CPU and single GPU training with *distributed_backend=dp*

Parameters

- **env** (`str`) – gym environment tag
- **gamma** (`float`) – discount factor
- **lr** (`float`) – learning rate
- **batch_size** (`int`) – size of minibatch pulled from the DataLoader
- **n_steps** (`int`) – number of states per discounted experience
- **entropy_beta** (`float`) – entropy coefficient
- **epoch_len** (`int`) – how many batches before pseudo epoch
- **num_batch_episodes** (`int`) – how many episodes to rollout for each batch of training
- **avg_reward_len** (`int`) – how many episodes to take into account when calculating the avg reward

_dataloader ()

Initialize the Replay Buffer dataset used for retrieving experiences

Return type `DataLoader`

static add_model_specific_args (*arg_parser*)
Adds arguments for DQN model

Note: These params are fine tuned for Pong env.

Parameters *arg_parser* – the current argument parser to add to

Return type `ArgumentParser`

Returns *arg_parser* with model specific args added

calc_qvals (*rewards*)

Calculate the discounted rewards of all rewards in list

Parameters *rewards* (`List[float]`) – list of rewards from latest batch

Return type `List[float]`

Returns list of discounted rewards

configure_optimizers ()

Initialize Adam optimizer

Return type `List[Optimizer]`

discount_rewards (*experiences*)

Calculates the discounted reward over N experiences

Parameters *experiences* (`Tuple[Experience]`) – Tuple of Experience

Return type `float`

Returns total discounted reward

forward (*x*)

Passes in a state *x* through the network and gets the *q*_values of each action as an output

Parameters *x* (`Tensor`) – environment state

Return type `Tensor`

Returns *q* values

get_device (*batch*)

Retrieve device currently being used by minibatch

Return type `str`

loss (*states*, *actions*, *scaled_rewards*)

Return type `Tensor`

train_batch ()

Contains the logic for generating a new batch of data to be passed to the DataLoader

Yields yields a tuple of Lists containing tensors for states, actions and rewards of the batch.

Return type `Tuple[List[Tensor], List[Tensor], List[Tensor]]`

train_dataloader ()

Get train loader

Return type `DataLoader`

training_step (*batch*, *_*)

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch recieved

Parameters

- **batch** (`Tuple[Tensor, Tensor]`) – current mini batch of replay data
- **_** – batch number, not used

Return type `OrderedDict`**Returns** Training loss and log metrics

```
pl_bolts.models.rl.reinforce_model.cli_main()
```

pl_bolts.models.rl.vanilla_policy_gradient_model module

```
class pl_bolts.models.rl.vanilla_policy_gradient_model.VanillaPolicyGradient (env,
                                                                              gamma=0.99,
                                                                              lr=0.01,
                                                                              batch_size=8,
                                                                              n_steps=10,
                                                                              avg_reward_len=10,
                                                                              en-
                                                                              tropy_beta=0.01,
                                                                              epoch_len=1000,
                                                                              **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [Vanilla Policy Gradient](#)

Paper authors: Richard S. Sutton, David McAllester, Satinder Singh, Yishay Mansour

Model implemented by:

- Donal Byrne <<https://github.com/djbyrne>>

Example

```
>>> from pl_bolts.models.rl.vanilla_policy_gradient_model import _
      ↪ VanillaPolicyGradient
      ...
>>> model = VanillaPolicyGradient("CartPole-v0")
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Note: This example is based on: https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On-Second-Edition/blob/master/Chapter11/04_cartpole_pg.py

Note: Currently only supports CPU and single GPU training with *distributed_backend=dp*

Parameters

- **env** (`str`) – gym environment tag
- **gamma** (`float`) – discount factor
- **lr** (`float`) – learning rate
- **batch_size** (`int`) – size of minibatch pulled from the DataLoader
- **batch_episodes** – how many episodes to rollout for each batch of training
- **entropy_beta** (`float`) – dictates the level of entropy per batch
- **avg_reward_len** (`int`) – how many episodes to take into account when calculating the avg reward
- **epoch_len** (`int`) – how many batches before pseudo epoch

`_dataloader()`

Initialize the Replay Buffer dataset used for retrieving experiences

Return type `DataLoader`

`static add_model_specific_args(arg_parser)`

Adds arguments for DQN model

Note: These params are fine tuned for Pong env.

Parameters **arg_parser** – the current argument parser to add to

Return type `ArgumentParser`

Returns `arg_parser` with model specific args added

`compute_returns(rewards)`

Calculate the discounted rewards of the batched rewards

Parameters **rewards** – list of batched rewards

Returns list of discounted rewards

`configure_optimizers()`

Initialize Adam optimizer

Return type `List[Optimizer]`

`forward(x)`

Passes in a state `x` through the network and gets the `q_values` of each action as an output

Parameters **x** (`Tensor`) – environment state

Return type `Tensor`

Returns `q values`

`get_device(batch)`

Retrieve device currently being used by minibatch

Return type `str`

`loss(states, actions, scaled_rewards)`

Calculates the loss for VPG

Parameters

- **states** – batched states
- **actions** – batch actions
- **scaled_rewards** – batched Q values

Return type `Tensor`**Returns** loss for the current batch**train_batch()**

Contains the logic for generating a new batch of data to be passed to the DataLoader

Return type `Tuple[List[Tensor], List[Tensor], List[Tensor]]`**Returns** yields a tuple of Lists containing tensors for states, actions and rewards of the batch.**train_dataloader()**

Get train loader

Return type `DataLoader`**training_step(batch, _)**

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch received

Parameters

- **batch** (`Tuple[Tensor, Tensor]`) – current mini batch of replay data
- **_** – batch number, not used

Return type `OrderedDict`**Returns** Training loss and log metrics`pl_bolts.models.rl.vanilla_policy_gradient_model.cli_main()`**pl_bolts.models.self_supervised package**

These models have been pre-trained using self-supervised learning. The models can also be used without pre-training and overwritten for your own research.

Here's an example for using these as pretrained models.

```
from pl_bolts.models.self_supervised import CPCV2

images = get_imagenet_batch()

# extract unsupervised representations
pretrained = CPCV2(pretrained=True)
representations = pretrained(images)

# use these in classification or any downstream task
classifications = classifier(representations)
```

```
class pl_bolts.models.self_supervised.AMDIM (datamodule='cifar10',          en-
                                             coder='amd_encoder',          con-
                                             trastive_task=torch.nn.Module,    im-
                                             age_channels=3,          image_height=32,
                                             encoder_feature_dim=320,      embed-
                                             ding_fx_dim=1280,   conv_block_depth=10,
                                             use_bn=False,          tclip=20.0,      learn-
                                             ing_rate=0.0002,          data_dir="",
                                             num_classes=10,          batch_size=200,
                                             num_workers=16, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [Augmented Multiscale Deep InfoMax \(AMDIM\)](#).

Paper authors: Philip Bachman, R Devon Hjelm, William Buchwalter.

Model implemented by: [William Falcon](#)

This code is adapted to Lightning using the original author repo ([the original repo](#)).

Example

```
>>> from pl_bolts.models.self_supervised import AMDIM
...
>>> model = AMDIM(encoder='resnet18')
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Parameters

- **datamodule** (`Union[str, LightningDataModule]`) – A LightningDatamodule
- **encoder** (`Union[str, Module, LightningModule]`) – an encoder string or model
- **image_channels** (`int`) – 3
- **image_height** (`int`) – pixels
- **encoder_feature_dim** (`int`) – Called *ndf* in the paper, this is the representation size for the encoder.
- **embedding_fx_dim** (`int`) – Output dim of the embedding function (*nrkhs* in the paper) (Reproducing Kernel Hilbert Spaces).
- **conv_block_depth** (`int`) – Depth of each encoder block,
- **use_bn** (`bool`) – If true will use batchnorm.
- **tclip** (`int`) – soft clipping non-linearity to the scores after computing the regularization term and before computing the log-softmax. This is the ‘second trick’ used in the paper
- **learning_rate** (`int`) – The learning rate
- **data_dir** (`str`) – Where to store data
- **num_classes** (`int`) – How many classes in the dataset
- **batch_size** (`int`) – The batch size

```
static add_model_specific_args (parent_parser)

configure_optimizers ()

forward (img_1, img_2)

init_encoder ()

train_dataloader ()

training_step (batch, batch_nb)

training_step_end (outputs)

val_dataloader ()

validation_epoch_end (outputs)

validation_step (batch, batch_nb)

class pl_bolts.models.self_supervised.BYOL (num_classes,          learning_rate=0.2,
                                           weight_decay=1.5e-06,   input_height=32,
                                           batch_size=32,          num_workers=0,
                                           warmup_epochs=10,       max_epochs=1000,
                                           **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [Bootstrap Your Own Latent \(BYOL\)](#)

Paper authors: Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre H. Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, Bilal Piot, Koray Kavukcuoglu, Rémi Munos, Michal Valko.

Model implemented by:

- [Annika Brundyn](#)

Warning: Work in progress. This implementation is still being verified.

TODOs:

- verify on CIFAR-10
- verify on STL-10
- pre-train on imagenet

Example:

```
model = BYOL(num_classes=10)

dm = CIFAR10DataModule(num_workers=0)
dm.train_transforms = SimCLRTrainDataTransform(32)
dm.val_transforms = SimCLREvalDataTransform(32)

trainer = pl.Trainer()
trainer.fit(model, datamodule=dm)
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```


CLI command:

```
# cifar10
python byol_module.py --gpus 1

# imagenet
python byol_module.py
    --gpus 8
    --dataset imagenet2012
    --data_dir /path/to/imagenet/
    --meta_dir /path/to/folder/with/meta.bin/
    --batch_size 32
```

Parameters

- **datamodule** – The datamodule
- **learning_rate** (*float*) – the learning rate
- **weight_decay** (*float*) – optimizer weight decay
- **input_height** (*int*) – image input height
- **batch_size** (*int*) – the batch size
- **num_workers** (*int*) – number of workers
- **warmup_epochs** (*int*) – num of epochs for scheduler warm up
- **max_epochs** (*int*) – max epochs for scheduler

static add_model_specific_args (*parent_parser*)

configure_optimizers ()

forward (*x*)

on_train_batch_end (*outputs, batch, batch_idx, dataloader_idx*)

Return type *None*

shared_step (*batch, batch_idx*)

training_step (*batch, batch_idx*)

validation_step (*batch, batch_idx*)

```
class pl_bolts.models.self_supervised.CPCV2 (encoder_name='cpc_encoder',
                                             patch_size=8, patch_overlap=4, on-
                                             line_ft=True, task='cpc', num_workers=4,
                                             num_classes=10, learning_rate=0.0001,
                                             pretrained=None, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Parameters

- **encoder_name** (*str*) – A string for any of the resnets in torchvision, or the original CPC encoder, or a custom nn.Module encoder
- **patch_size** (*int*) – How big to make the image patches
- **patch_overlap** (*int*) – How much overlap each patch should have
- **online_ft** (*bool*) – If True, enables a 1024-unit MLP to fine-tune online
- **task** (*str*) – Which self-supervised task to use ('cpc', 'amdin', etc...)

- **num_workers** (*int*) – number of dataloader workers
- **num_classes** (*int*) – number of classes
- **learning_rate** (*float*) – learning rate
- **pretrained** (*Optional[str]*) – If true, will use the weights pretrained (using CPC) on Imagenet

_CPCV2__compute_final_nb_c (*patch_size*)

_CPCV2__recover_z_shape (*Z, b*)

static add_model_specific_args (*parent_parser*)

configure_optimizers ()

forward (*img_1*)

init_encoder ()

load_pretrained (*encoder_name*)

shared_step (*batch*)

training_step (*batch, batch_nb*)

validation_step (*batch, batch_nb*)

```
class pl_bolts.models.self_supervised.SSLEvaluator (n_input, n_classes,
                                                    n_hidden=512, p=0.1)
```

Bases: `torch.nn.Module`

forward (*x*)

```
class pl_bolts.models.self_supervised.MocoV2 (base_encoder='resnet18', emb_dim=128,
                                              num_negatives=65536, en-
                                              coder_momentum=0.999, soft-
                                              max_temperature=0.07, learn-
                                              ing_rate=0.03, momentum=0.9,
                                              weight_decay=0.0001, data_dir='./',
                                              batch_size=256, use_mlp=False,
                                              num_workers=8, *args, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [Moco](#)

Paper authors: Xinlei Chen, Haoqi Fan, Ross Girshick, Kaiming He.

Code adapted from [facebookresearch/moco](#) to Lightning by:

- [William Falcon](#)

Example:: `from pl_bolts.models.self_supervised import MocoV2 model = MocoV2() trainer = Trainer() trainer.fit(model)`

CLI command:

```
# cifar10
python moco2_module.py --gpus 1

# imagenet
python moco2_module.py
    --gpus 8
```

(continues on next page)

(continued from previous page)

```
--dataset imagenet2012
--data_dir /path/to/imagenet/
--meta_dir /path/to/folder/with/meta.bin/
--batch_size 32
```

Parameters

- **base_encoder** (`Union[str, Module]`) – torchvision model name or torch.nn.Module
- **emb_dim** (`int`) – feature dimension (default: 128)
- **num_negatives** (`int`) – queue size; number of negative keys (default: 65536)
- **encoder_momentum** (`float`) – moco momentum of updating key encoder (default: 0.999)
- **softmax_temperature** (`float`) – softmax temperature (default: 0.07)
- **learning_rate** (`float`) – the learning rate
- **momentum** (`float`) – optimizer momentum
- **weight_decay** (`float`) – optimizer weight decay
- **datamodule** – the DataModule (train, val, test dataloaders)
- **data_dir** (`str`) – the directory to store data
- **batch_size** (`int`) – batch size
- **use_mlp** (`bool`) – add an mlp to the encoders
- **num_workers** (`int`) – workers for the loaders

_batch_shuffle_ddp (*x*)

Batch shuffle, for making use of BatchNorm. * **Only support DistributedDataParallel (DDP) model.** *

_batch_unshuffle_ddp (*x, idx_unshuffle*)

Undo batch shuffle. * **Only support DistributedDataParallel (DDP) model.** *

_dequeue_and_enqueue (*keys*)

_momentum_update_key_encoder ()

Momentum update of the key encoder

static add_model_specific_args (*parent_parser*)

configure_optimizers ()

forward (*img_q, img_k*)

Input: *img_q*: a batch of query images *img_k*: a batch of key images

Output: logits, targets

init_encoders (*base_encoder*)

Override to add your own encoders

training_step (*batch, batch_idx*)

validation_epoch_end (*outputs*)

validation_step (*batch, batch_idx*)

```
class pl_bolts.models.self_supervised.SimCLR(gpus, num_samples, batch_size,
dataset, num_nodes=1, arch='resnet50',
hidden_mlp=2048, feat_dim=128,
warmup_epochs=10, max_epochs=100,
temperature=0.1, first_conv=True,
maxpool1=True, optimizer='adam',
lars_wrapper=True, exclude_bn_bias=False, start_lr=0.0,
learning_rate=0.001, final_lr=0.0,
weight_decay=1e-06, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Parameters

- **batch_size** (`int`) – the batch size
- **num_samples** (`int`) – num samples in the dataset
- **warmup_epochs** (`int`) – epochs to warmup the lr for
- **lr** – the optimizer learning rate
- **opt_weight_decay** – the optimizer weight decay
- **loss_temperature** – the loss temperature

static add_model_specific_args (*parent_parser*)

configure_optimizers ()

exclude_from_wt_decay (*named_params*, *weight_decay*, *skip_list*=['bias', 'bn'])

forward (*x*)

init_model ()

nt_xent_loss (*out_1*, *out_2*, *temperature*, *eps*=1e-06)

assume out_1 and out_2 are normalized out_1: [batch_size, dim] out_2: [batch_size, dim]

optimizer_step (*epoch*=None, *batch_idx*=None, *optimizer*=None, *optimizer_idx*=None, *optimizer_closure*=None, *on_tpu*=None, *using_native_amp*=None, *using_lbfgs*=None)

Return type None

shared_step (*batch*)

training_step (*batch*, *batch_idx*)

validation_step (*batch*, *batch_idx*)

```
class pl_bolts.models.self_supervised.SimSiam(gpus, num_samples, batch_size,
dataset, num_nodes=1, arch='resnet50',
hidden_mlp=2048, feat_dim=128,
warmup_epochs=10, max_epochs=100,
temperature=0.1, first_conv=True,
maxpool1=True, optimizer='adam',
lars_wrapper=True, exclude_bn_bias=False, start_lr=0.0,
learning_rate=0.001, final_lr=0.0,
weight_decay=1e-06, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of Exploring Simple Siamese Representation Learning (SimSiam)

Paper authors: Xinlei Chen, Kaiming He.

Model implemented by:

- [Zvi Lapp](#)

Warning: Work in progress. This implementation is still being verified.

TODOs:

- verify on CIFAR-10
- verify on STL-10
- pre-train on imagenet

Example:

```
model = SimSiam()

dm = CIFAR10DataModule(num_workers=0)
dm.train_transforms = SimCLRTrainDataTransform(32)
dm.val_transforms = SimCLREvalDataTransform(32)

trainer = pl.Trainer()
trainer.fit(model, datamodule=dm)
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

CLI command:

```
# cifar10
python simsiam_module.py --gpus 1

# imagenet
python simsiam_module.py
  --gpus 8
  --dataset imagenet2012
  --data_dir /path/to/imagenet/
  --meta_dir /path/to/folder/with/meta.bin/
  --batch_size 32
```

Parameters

- **datamodule** – The datamodule
- **learning_rate** (*float*) – the learning rate
- **weight_decay** (*float*) – optimizer weight decay
- **input_height** – image input height
- **batch_size** (*int*) – the batch size
- **num_workers** – number of workers
- **warmup_epochs** (*int*) – num of epochs for scheduler warm up

- `max_epochs` (`int`) – max epochs for scheduler

`static add_model_specific_args` (`parent_parser`)

`configure_optimizers` ()

`cosine_similarity` (`a`, `b`)

`exclude_from_wt_decay` (`named_params`, `weight_decay`, `skip_list=['bias', 'bn']`)

`forward` (`x`)

`init_model` ()

`optimizer_step` (`epoch`, `batch_idx`, `optimizer`, `optimizer_idx`, `optimizer_closure=None`,
`on_tpu=False`, `using_native_amp=False`, `using_lbfgs=False`)

Return type `None`

`training_step` (`batch`, `batch_idx`)

`validation_step` (`batch`, `batch_idx`)

```
class pl_bolts.models.self_supervised.SSLFineTuner (backbone,      in_features=2048,
                                                    num_classes=1000,
                                                    epochs=100,  hidden_dim=None,
                                                    dropout=0.0,  learning_rate=0.1,
                                                    weight_decay=1e-06,
                                                    nesterov=False,      sched-
                                                    uler_type='cosine',      de-
                                                    cay_epochs=[60,      80],
                                                    gamma=0.1, final_lr=0.0)
```

Bases: `pytorch_lightning.LightningModule`

Finetunes a self-supervised learning backbone using the standard evaluation protocol of a singler layer MLP with 1024 units

Example:

```
from pl_bolts.utils.self_supervised import SSLFineTuner
from pl_bolts.models.self_supervised import CPCV2
from pl_bolts.datamodules import CIFAR10DataModule
from pl_bolts.models.self_supervised.cpc.transforms import _
↳CPCEvalTransformsCIFAR10,
↳CPCTrainTransformsCIFAR10

# pretrained model
backbone = CPCV2.load_from_checkpoint(PATH, strict=False)

# dataset + transforms
dm = CIFAR10DataModule(data_dir='.')
dm.train_transforms = CPCTrainTransformsCIFAR10()
dm.val_transforms = CPCEvalTransformsCIFAR10()

# finetuner
finetuner = SSLFineTuner(backbone, in_features=backbone.z_dim, num_
↳classes=backbone.num_classes)

# train
trainer = pl.Trainer()
trainer.fit(finetuner, dm)
```

(continues on next page)

(continued from previous page)

```
# test
trainer.test(datamodule=dm)
```

Parameters

- **backbone** (`Module`) – a pretrained model
- **in_features** (`int`) – feature dim of backbone outputs
- **num_classes** (`int`) – classes of the dataset
- **hidden_dim** (`Optional[int]`) – dim of the MLP (1024 default used in self-supervised literature)

configure_optimizers ()

on_train_epoch_start ()

Return type `None`

shared_step (*batch*)

test_step (*batch*, *batch_idx*)

training_step (*batch*, *batch_idx*)

validation_step (*batch*, *batch_idx*)

```
class pl_bolts.models.self_supervised.SwAV(gpus,          num_samples,      batch_size,
                                             dataset,          num_nodes=1,
                                             arch='resnet50',    hidden_mlp=2048,
                                             feat_dim=128,        warmup_epochs=10,
                                             max_epochs=100,      nmb_prototypes=3000,
                                             freeze_prototypes_epochs=1, temperature=0.1,
                                             sinkhorn_iterations=3,
                                             queue_length=0,     queue_path='queue',
                                             epoch_queue_starts=15,
                                             crops_for_assign=[0, 1], nmb_crops=[2,
6], first_conv=True, maxpool1=True,
                                             optimizer='adam',   lars_wrapper=True,
                                             exclude_bn_bias=False, start_lr=0.0,
                                             learning_rate=0.001, final_lr=0.0,
                                             weight_decay=1e-06,  epsilon=0.05,
                                             **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Parameters

- **gpus** (`int`) – number of gpus per node used in training, passed to SwAV module to manage the queue and select distributed sinkhorn
- **num_nodes** (`int`) – number of nodes to train on
- **num_samples** (`int`) – number of image samples used for training
- **batch_size** (`int`) – batch size per GPU in ddp
- **dataset** (`str`) – dataset being used for train/val
- **arch** (`str`) – encoder architecture used for pre-training

- **hidden_mlp** (*int*) – hidden layer of non-linear projection head, set to 0 to use a linear projection head
- **feat_dim** (*int*) – output dim of the projection head
- **warmup_epochs** (*int*) – apply linear warmup for this many epochs
- **max_epochs** (*int*) – epoch count for pre-training
- **nmb_prototypes** (*int*) – count of prototype vectors
- **freeze_prototypes_epochs** (*int*) – epoch till which gradients of prototype layer are frozen
- **temperature** (*float*) – loss temperature
- **sinkhorn_iterations** (*int*) – iterations for sinkhorn normalization
- **queue_length** (*int*) – set queue when batch size is small, must be divisible by total batch-size (i.e. `total_gpus * batch_size`), set to 0 to remove the queue
- **queue_path** (*str*) – folder within the logs directory
- **epoch_queue_starts** (*int*) – start using the queue after this epoch
- **crops_for_assign** (*list*) – list of crop ids for computing assignment
- **nmb_crops** (*list*) – number of global and local crops, ex: [2, 6]
- **first_conv** (*bool*) – keep first conv same as the original resnet architecture, if set to false it is replace by a kernel 3, stride 1 conv (cifar-10)
- **maxpool1** (*bool*) – keep first maxpool layer same as the original resnet architecture, if set to false, first maxpool is turned off (cifar10, maybe stl10)
- **optimizer** (*str*) – optimizer to use
- **lars_wrapper** (*bool*) – use LARS wrapper over the optimizer
- **exclude_bn_bias** (*bool*) – exclude batchnorm and bias layers from weight decay in optimizers
- **start_lr** (*float*) – starting lr for linear warmup
- **learning_rate** (*float*) – learning rate
- **final_lr** (*float*) – float = final learning rate for cosine weight decay
- **weight_decay** (*float*) – weight decay for optimizer
- **epsilon** (*float*) – epsilon val for swav assignments

```
static add_model_specific_args (parent_parser)
configure_optimizers ()
distributed_sinkhorn (Q, nmb_iters)
exclude_from_wt_decay (named_params, weight_decay, skip_list=['bias', 'bn'])
forward (x)
init_model ()
on_after_backward ()
on_train_epoch_end (outputs)
```

Return type None


```

on_train_epoch_start ()

optimizer_step (epoch=None, batch_idx=None, optimizer=None, optimizer_idx=None,
                optimizer_closure=None, on_tpu=None, using_native_amp=None, us-
                ing_lbfgs=None)

    Return type None

setup (stage)

shared_step (batch)

sinkhorn (Q, nmb_iters)

training_step (batch, batch_idx)

validation_step (batch, batch_idx)

```

Subpackages

pl_bolts.models.self_supervised.amdim package

```

class pl_bolts.models.self_supervised.amdim.AMDIM (datamodule='cifar10', en-
                                                    coder='amdim_encoder', con-
                                                    trastive_task=torch.nn.Module, im-
                                                    age_channels=3, image_height=32,
                                                    encoder_feature_dim=320,
                                                    embedding_fx_dim=1280,
                                                    conv_block_depth=10,
                                                    use_bn=False, tclip=20.0, learn-
                                                    ing_rate=0.0002, data_dir="",
                                                    num_classes=10, batch_size=200,
                                                    num_workers=16, **kwargs)

```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [Augmented Multiscale Deep InfoMax \(AMDIM\)](#).

Paper authors: Philip Bachman, R Devon Hjelm, William Buchwalter.

Model implemented by: [William Falcon](#)

This code is adapted to Lightning using the original author repo ([the original repo](#)).

Example

```

>>> from pl_bolts.models.self_supervised import AMDIM
...
>>> model = AMDIM(encoder='resnet18')

```

Train:

```

trainer = Trainer()
trainer.fit(model)

```

Parameters

- **datamodule** (`Union[str, LightningDataModule]`) – A LightningDatamodule
- **encoder** (`Union[str, Module, LightningModule]`) – an encoder string or model

- **image_channels** (*int*) – 3
- **image_height** (*int*) – pixels
- **encoder_feature_dim** (*int*) – Called *ndf* in the paper, this is the representation size for the encoder.
- **embedding_fx_dim** (*int*) – Output dim of the embedding function (*nrkhs* in the paper) (Reproducing Kernel Hilbert Spaces).
- **conv_block_depth** (*int*) – Depth of each encoder block,
- **use_bn** (*bool*) – If true will use batchnorm.
- **tclip** (*int*) – soft clipping non-linearity to the scores after computing the regularization term and before computing the log-softmax. This is the ‘second trick’ used in the paper
- **learning_rate** (*int*) – The learning rate
- **data_dir** (*str*) – Where to store data
- **num_classes** (*int*) – How many classes in the dataset
- **batch_size** (*int*) – The batch size

```
static add_model_specific_args (parent_parser)
configure_optimizers ()
forward (img_1, img_2)
init_encoder ()
train_dataloader ()
training_step (batch, batch_nb)
training_step_end (outputs)
val_dataloader ()
validation_epoch_end (outputs)
validation_step (batch, batch_nb)

class pl_bolts.models.self_supervised.amdim.AMDIMEncoder (dummy_batch,
                                                         num_channels=3,  en-
                                                         coder_feature_dim=64,
                                                         embedding_fx_dim=512,
                                                         conv_block_depth=3,
                                                         encoder_size=32,
                                                         use_bn=False)

Bases: torch.nn.Module
__config_modules (x, output_widths, n_rkhs, use_bn)
    Configure the modules for extracting fake rkhs embeddings for infomax.
__forward_acts (x)
    Return activations from all layers.
forward (x)
init_weights (init_scale=1.0)
    Run custom weight init for modules...
```

class `pl_bolts.models.self_supervised.amdim.AMDIMEvalTransformsCIFAR10`

Bases: `object`

Transforms applied to AMDIM

Transforms:

```
transforms.ToTensor(),
normalize
```

Example:

```
x = torch.rand(5, 3, 32, 32)

transform = AMDIMEvalTransformsCIFAR10()
(view1, view2) = transform(x)
```

`__call__`(*inp*)

Call self as a function.

class `pl_bolts.models.self_supervised.amdim.AMDIMEvalTransformsImageNet128` (*height=128*)

Bases: `object`

Transforms applied to AMDIM

Transforms:

```
transforms.Resize(height + 6, interpolation=3),
transforms.CenterCrop(height),
transforms.ToTensor(),
normalize
```

Example:

```
x = torch.rand(5, 3, 128, 128)

transform = AMDIMEvalTransformsImageNet128()
view1 = transform(x)
```

`__call__`(*inp*)

Call self as a function.

class `pl_bolts.models.self_supervised.amdim.AMDIMEvalTransformsSTL10` (*height=64*)

Bases: `object`

Transforms applied to AMDIM

Transforms:

```
transforms.Resize(height + 6, interpolation=3),
transforms.CenterCrop(height),
transforms.ToTensor(),
normalize
```

Example:

```
x = torch.rand(5, 3, 64, 64)

transform = AMDIMTrainTransformsSTL10()
view1 = transform(x)
```

`__call__(inp)`
Call self as a function.

class `pl_bolts.models.self_supervised.amdim.AMDIMTrainTransformsCIFAR10`

Bases: `object`

Transforms applied to AMDIM

Transforms:

```
img_jitter,  
col_jitter,  
rnd_gray,  
transforms.ToTensor(),  
normalize
```

Example:

```
x = torch.rand(5, 3, 32, 32)  
  
transform = AMDIMTrainTransformsCIFAR10()  
(view1, view2) = transform(x)
```

`__call__(inp)`
Call self as a function.

class `pl_bolts.models.self_supervised.amdim.AMDIMTrainTransformsImageNet128 (height=128)`

Bases: `object`

Transforms applied to AMDIM

Transforms:

```
img_jitter,  
col_jitter,  
rnd_gray,  
transforms.ToTensor(),  
normalize
```

Example:

```
x = torch.rand(5, 3, 128, 128)  
  
transform = AMDIMTrainTransformsSTL10()  
(view1, view2) = transform(x)
```

`__call__(inp)`
Call self as a function.

class `pl_bolts.models.self_supervised.amdim.AMDIMTrainTransformsSTL10 (height=64)`

Bases: `object`

Transforms applied to AMDIM

Transforms:

```
img_jitter,  
col_jitter,  
rnd_gray,  
transforms.ToTensor(),  
normalize
```

Example:

```
x = torch.rand(5, 3, 64, 64)

transform = AMDIMTrainTransformsSTL10()
(view1, view2) = transform(x)
```

`__call__` (*inp*)
Call self as a function.

Submodules

`pl_bolts.models.self_supervised.amdim.amdim_module` module

```
class pl_bolts.models.self_supervised.amdim.amdim_module.AMDIM(datamodule='cifar10',
    en-
    coder='amdim_encoder',
    con-
    trastive_task=torch.nn.Module,
    im-
    age_channels=3,
    im-
    age_height=32,
    en-
    coder_feature_dim=320,
    embed-
    ding_fx_dim=1280,
    conv_block_depth=10,
    use_bn=False,
    tclip=20.0,
    learn-
    ing_rate=0.0002,
    data_dir="",
    num_classes=10,
    batch_size=200,
    num_workers=16,
    **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [Augmented Multiscale Deep InfoMax \(AMDIM\)](#).

Paper authors: Philip Bachman, R Devon Hjelm, William Buchwalter.

Model implemented by: [William Falcon](#)

This code is adapted to Lightning using the original author repo ([the original repo](#)).

Example

```
>>> from pl_bolts.models.self_supervised import AMDIM
...
>>> model = AMDIM(encoder='resnet18')
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

Parameters

- **datamodule** (`Union[str, LightningDataModule]`) – A LightningDatamodule
- **encoder** (`Union[str, Module, LightningModule]`) – an encoder string or model
- **image_channels** (`int`) – 3
- **image_height** (`int`) – pixels
- **encoder_feature_dim** (`int`) – Called *ndf* in the paper, this is the representation size for the encoder.
- **embedding_fx_dim** (`int`) – Output dim of the embedding function (*nrkhs* in the paper) (Reproducing Kernel Hilbert Spaces).
- **conv_block_depth** (`int`) – Depth of each encoder block,
- **use_bn** (`bool`) – If true will use batchnorm.
- **tclip** (`int`) – soft clipping non-linearity to the scores after computing the regularization term and before computing the log-softmax. This is the ‘second trick’ used in the paper
- **learning_rate** (`int`) – The learning rate
- **data_dir** (`str`) – Where to store data
- **num_classes** (`int`) – How many classes in the dataset
- **batch_size** (`int`) – The batch size

```
static add_model_specific_args (parent_parser)
configure_optimizers ()
forward (img_1, img_2)
init_encoder ()
train_dataloader ()
training_step (batch, batch_nb)
training_step_end (outputs)
val_dataloader ()
validation_epoch_end (outputs)
validation_step (batch, batch_nb)

pl_bolts.models.self_supervised.amdim.amdim_module.cli_main ()
```

pl_bolts.models.self_supervised.amdim.datasets module

```

class pl_bolts.models.self_supervised.amdim.datasets.AMDIMPatchesPretraining
    Bases: object
    """ For pretraining we use the train transform for both train and val.
    static cifar10 (dataset_root, patch_size, patch_overlap, split='train')
    static imagenet (dataset_root, nb_classes, patch_size, patch_overlap, split='train')
    static stl (dataset_root, patch_size, patch_overlap, split=None)

class pl_bolts.models.self_supervised.amdim.datasets.AMDIMPretraining
    Bases: object
    """ For pretraining we use the train transform for both train and val.
    static cifar10 (dataset_root, split='train')
    static cifar10_tiny (dataset_root, split='train')
    static get_dataset (datamodule, data_dir, split='train', **kwargs)
    static imagenet (dataset_root, nb_classes, split='train')
    static stl (dataset_root, split=None)

```

pl_bolts.models.self_supervised.amdim.networks module

```

class pl_bolts.models.self_supervised.amdim.networks.AMDIMEncoder (dummy_batch,
                                                                    num_channels=3,
                                                                    en-
                                                                    coder_feature_dim=64,
                                                                    embed-
                                                                    ding_fx_dim=512,
                                                                    conv_block_depth=3,
                                                                    en-
                                                                    coder_size=32,
                                                                    use_bn=False)

    Bases: torch.nn.Module
    _config_modules (x, output_widths, n_rkhs, use_bn)
        Configure the modules for extracting fake rkhs embeddings for infomax.
    _forward_acts (x)
        Return activations from all layers.
    forward (x)
    init_weights (init_scale=1.0)
        Run custom weight init for modules...

class pl_bolts.models.self_supervised.amdim.networks.Conv3x3 (n_in,          n_out,
                                                                n_kern,
                                                                n_stride,  n_pad,
                                                                use_bn=True,
                                                                pad_mode='constant')

    Bases: torch.nn.Module
    forward (x)

```

```
class pl_bolts.models.self_supervised.amdim.networks.ConvResBlock (n_in, n_out,  
                                                                width,  
                                                                stride,  
                                                                pad, depth,  
                                                                use_bn)  
  
    Bases: torch.nn.Module  
  
    forward (x)  
  
    init_weights (init_scale=1.0)  
        Do a fixup-ish init for each ConvResNxN in this block.  
  
class pl_bolts.models.self_supervised.amdim.networks.ConvResNxN (n_in, n_out,  
                                                                width,  
                                                                stride, pad,  
                                                                use_bn=False)  
  
    Bases: torch.nn.Module  
  
    forward (x)  
  
    init_weights (init_scale=1.0)  
  
class pl_bolts.models.self_supervised.amdim.networks.FakeRKHSConvNet (n_input,  
                                                                n_output,  
                                                                use_bn=False)  
  
    Bases: torch.nn.Module  
  
    forward (x)  
  
    init_weights (init_scale=1.0)  
  
class pl_bolts.models.self_supervised.amdim.networks.MaybeBatchNorm2d (n_fir,  
                                                                affine,  
                                                                use_bn)  
  
    Bases: torch.nn.Module  
  
    forward (x)  
  
class pl_bolts.models.self_supervised.amdim.networks.NopNet (norm_dim=None)  
    Bases: torch.nn.Module  
  
    forward (x)
```

pl_bolts.models.self_supervised.amdim.transforms module

```
class pl_bolts.models.self_supervised.amdim.transforms.AMDIMEvalTransformsCIFAR10  
    Bases: object  
  
    Transforms applied to AMDIM  
  
    Transforms:
```

```
transforms.ToTensor(),  
normalize
```

Example:

```
x = torch.rand(5, 3, 32, 32)  
  
transform = AMDIMEvalTransformsCIFAR10()  
(view1, view2) = transform(x)
```


`__call__(inp)`
Call self as a function.

class `pl_bolts.models.self_supervised.amdim.transforms.AMDIMEvalTransformsImageNet128` (*height*)

Bases: `object`

Transforms applied to AMDIM

Transforms:

```
transforms.Resize(height + 6, interpolation=3),
transforms.CenterCrop(height),
transforms.ToTensor(),
normalize
```

Example:

```
x = torch.rand(5, 3, 128, 128)

transform = AMDIMEvalTransformsImageNet128()
view1 = transform(x)
```

`__call__(inp)`
Call self as a function.

class `pl_bolts.models.self_supervised.amdim.transforms.AMDIMEvalTransformsSTL10` (*height=64*)

Bases: `object`

Transforms applied to AMDIM

Transforms:

```
transforms.Resize(height + 6, interpolation=3),
transforms.CenterCrop(height),
transforms.ToTensor(),
normalize
```

Example:

```
x = torch.rand(5, 3, 64, 64)

transform = AMDIMTrainTransformsSTL10()
view1 = transform(x)
```

`__call__(inp)`
Call self as a function.

class `pl_bolts.models.self_supervised.amdim.transforms.AMDIMTrainTransformsCIFAR10`

Bases: `object`

Transforms applied to AMDIM

Transforms:

```
img_jitter,
col_jitter,
rnd_gray,
transforms.ToTensor(),
normalize
```

Example:

```
x = torch.rand(5, 3, 32, 32)

transform = AMDIMTrainTransformsCIFAR10()
(view1, view2) = transform(x)
```

__call__(*inp*)
Call self as a function.

class pl_bolts.models.self_supervised.amdim.transforms.**AMDIMTrainTransformsImageNet128** (*height=128*)
Bases: `object`

Transforms applied to AMDIM

Transforms:

```
img_jitter,
col_jitter,
rnd_gray,
transforms.ToTensor(),
normalize
```

Example:

```
x = torch.rand(5, 3, 128, 128)

transform = AMDIMTrainTransformsSTL10()
(view1, view2) = transform(x)
```

__call__(*inp*)
Call self as a function.

class pl_bolts.models.self_supervised.amdim.transforms.**AMDIMTrainTransformsSTL10** (*height=64*)
Bases: `object`

Transforms applied to AMDIM

Transforms:

```
img_jitter,
col_jitter,
rnd_gray,
transforms.ToTensor(),
normalize
```

Example:

```
x = torch.rand(5, 3, 64, 64)

transform = AMDIMTrainTransformsSTL10()
(view1, view2) = transform(x)
```

__call__(*inp*)
Call self as a function.

pl_bolts.models.self_supervised.byol package

Submodules

pl_bolts.models.self_supervised.byol.byol_module module

```
class pl_bolts.models.self_supervised.byol.byol_module.BYOL(num_classes, learning_rate=0.2,
                                                             weight_decay=1.5e-06, input_height=32,
                                                             batch_size=32, num_workers=0,
                                                             warmup_epochs=10, max_epochs=1000,
                                                             **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [Bootstrap Your Own Latent \(BYOL\)](#)

Paper authors: Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre H. Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, Bilal Piot, Koray Kavukcuoglu, Rémi Munos, Michal Valko.

Model implemented by:

- [Annika Brundyn](#)

Warning: Work in progress. This implementation is still being verified.

TODOs:

- verify on CIFAR-10
- verify on STL-10
- pre-train on imagenet

Example:

```
model = BYOL(num_classes=10)

dm = CIFAR10DataModule(num_workers=0)
dm.train_transforms = SimCLRTrainDataTransform(32)
dm.val_transforms = SimCLREvalDataTransform(32)

trainer = pl.Trainer()
trainer.fit(model, datamodule=dm)
```

Train:

```
trainer = Trainer()
trainer.fit(model)
```

CLI command:

```
# cifar10
python byol_module.py --gpus 1

# imagenet
python byol_module.py
    --gpus 8
    --dataset imagenet2012
    --data_dir /path/to/imagenet/
    --meta_dir /path/to/folder/with/meta.bin/
    --batch_size 32
```

Parameters

- **datamodule** – The datamodule
- **learning_rate** (*float*) – the learning rate
- **weight_decay** (*float*) – optimizer weight decay
- **input_height** (*int*) – image input height
- **batch_size** (*int*) – the batch size
- **num_workers** (*int*) – number of workers
- **warmup_epochs** (*int*) – num of epochs for scheduler warm up
- **max_epochs** (*int*) – max epochs for scheduler

static add_model_specific_args (*parent_parser*)

configure_optimizers ()

forward (*x*)

on_train_batch_end (*outputs, batch, batch_idx, dataloader_idx*)

Return type *None*

shared_step (*batch, batch_idx*)

training_step (*batch, batch_idx*)

validation_step (*batch, batch_idx*)

`pl_bolts.models.self_supervised.byol.byol_module.cli_main()`

`pl_bolts.models.self_supervised.byol.models` module

class `pl_bolts.models.self_supervised.byol.models.MLP` (*input_dim=2048, den_size=4096, put_dim=256*) *hid-out-*

Bases: `torch.nn.Module`

forward (*x*)

class `pl_bolts.models.self_supervised.byol.models.SiameseArm` (*encoder=None*)

Bases: `torch.nn.Module`

forward (*x*)

pl_bolts.models.self_supervised.cpc package

```
class pl_bolts.models.self_supervised.cpc.CPCV2(encoder_name='cpc_encoder',
                                              patch_size=8, patch_overlap=4,
                                              online_ft=True, task='cpc',
                                              num_workers=4, num_classes=10,
                                              learning_rate=0.0001, pre-
                                              trained=None, **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Parameters

- **encoder_name** (`str`) – A string for any of the resnets in torchvision, or the original CPC encoder, or a custom nn.Module encoder
- **patch_size** (`int`) – How big to make the image patches
- **patch_overlap** (`int`) – How much overlap each patch should have
- **online_ft** (`bool`) – If True, enables a 1024-unit MLP to fine-tune online
- **task** (`str`) – Which self-supervised task to use ('cpc', 'amdin', etc...)
- **num_workers** (`int`) – number of dataloader workers
- **num_classes** (`int`) – number of classes
- **learning_rate** (`float`) – learning rate
- **pretrained** (`Optional[str]`) – If true, will use the weights pretrained (using CPC) on Imagenet

```
_CPCV2__compute_final_nb_c (patch_size)
_CPCV2__recover_z_shape (Z, b)
static add_model_specific_args (parent_parser)
configure_optimizers ()
forward (img_1)
init_encoder ()
load_pretrained (encoder_name)
shared_step (batch)
training_step (batch, batch_nb)
validation_step (batch, batch_nb)
```

```
class pl_bolts.models.self_supervised.cpc.CPCEvalTransformsCIFAR10(patch_size=8,
                                                                    over-
                                                                    lap=4)
```

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=overlap)
```

Example:

```
# in a regular dataset
CIFAR10(..., transforms=CPCEvalTransformsCIFAR10())

# in a DataModule
module = CIFAR10DataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
    ↪transforms=CPCEvalTransformsCIFAR10())
```

Parameters

- **patch_size**(int) – size of patches when cutting up the image into overlapping patches
- **overlap**(int) – how much to overlap patches

`__call__`(inp)

Call self as a function.

```
class pl_bolts.models.self_supervised.cpc.CPCEvalTransformsImageNet128(patch_size=32,
                                                                    over-
                                                                    lap=16)
```

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=patch_size // 2)
```

Example:

```
# in a regular dataset
Imagenet(..., transforms=CPCEvalTransformsImageNet128())

# in a DataModule
module = ImagenetDataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
    ↪transforms=CPCEvalTransformsImageNet128())
```

Parameters

- **patch_size**(int) – size of patches when cutting up the image into overlapping patches
- **overlap**(int) – how much to overlap patches

`__call__`(inp)

Call self as a function.

```
class pl_bolts.models.self_supervised.cpc.CPCEvalTransformsSTL10(patch_size=16,
                                                                    overlap=8)
```

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=patch_size // 2)
```

Example:

```
# in a regular dataset
STL10(..., transforms=CPCEvalTransformsSTL10())

# in a DataModule
module = STL10DataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
↳ transforms=CPCEvalTransformsSTL10())
```

Parameters

- **patch_size** (*int*) – size of patches when cutting up the image into overlapping patches
- **overlap** (*int*) – how much to overlap patches

`__call__` (*inp*)

Call self as a function.

```
class pl_bolts.models.self_supervised.cpc.CPCTrainTransformsCIFAR10(patch_size=8,
                                                                    over-
                                                                    lap=4)
```

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
img_jitter
col_jitter
rnd_gray
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=patch_size // 2)
```

Example:

```
# in a regular dataset
CIFAR10(..., transforms=CPCTrainTransformsCIFAR10())

# in a DataModule
module = CIFAR10DataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
↳ transforms=CPCTrainTransformsCIFAR10())
```

Parameters

- **patch_size** – size of patches when cutting up the image into overlapping patches
- **overlap** – how much to overlap patches

`__call__` (*inp*)

Call self as a function.

```
class pl_bolts.models.self_supervised.cpc.CPCTrainTransformsImageNet128(patch_size=32,
                                                                    over-
                                                                    lap=16)
```

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=patch_size // 2)
```

Example:

```
# in a regular dataset
Imagenet(..., transforms=CPCTrainTransformsImageNet128())

# in a DataModule
module = ImagenetDataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
↳ transforms=CPCTrainTransformsImageNet128())
```

Parameters

- **patch_size** (`int`) – size of patches when cutting up the image into overlapping patches
- **overlap** (`int`) – how much to overlap patches

`__call__` (*inp*)

Call self as a function.

```
class pl_bolts.models.self_supervised.cpc.CPCTrainTransformsSTL10(patch_size=16,
                                                                    overlap=8)
```

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
img_jitter
col_jitter
rnd_gray
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=patch_size // 2)
```

Example:

```
# in a regular dataset
STL10(..., transforms=CPCTrainTransformsSTL10())

# in a DataModule
module = STL10DataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
↳ transforms=CPCTrainTransformsSTL10())
```

Parameters

- **patch_size** (*int*) – size of patches when cutting up the image into overlapping patches
- **overlap** (*int*) – how much to overlap patches

`__call__` (*inp*)

Call self as a function.

`pl_bolts.models.self_supervised.cpc.cpc_resnet50` (*sample_batch*, ***kwargs*)

`pl_bolts.models.self_supervised.cpc.cpc_resnet101` (*sample_batch*, ***kwargs*)

Submodules

`pl_bolts.models.self_supervised.cpc.cpc_finetuner` module

`pl_bolts.models.self_supervised.cpc.cpc_finetuner.cli_main` ()

`pl_bolts.models.self_supervised.cpc.cpc_module` module

CPC V2

```
class pl_bolts.models.self_supervised.cpc.cpc_module.CPCV2 (encoder_name='cpc_encoder',
                                                           patch_size=8,
                                                           patch_overlap=4,
                                                           online_ft=True,
                                                           task='cpc',
                                                           num_workers=4,
                                                           num_classes=10,
                                                           learning_rate=0.0001,
                                                           pretrained=None,
                                                           **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Parameters

- **encoder_name** (*str*) – A string for any of the resnets in torchvision, or the original CPC encoder, or a custom nn.Module encoder
- **patch_size** (*int*) – How big to make the image patches
- **patch_overlap** (*int*) – How much overlap each patch should have
- **online_ft** (*bool*) – If True, enables a 1024-unit MLP to fine-tune online
- **task** (*str*) – Which self-supervised task to use ('cpc', 'amdin', etc...)
- **num_workers** (*int*) – number of dataloader workers
- **num_classes** (*int*) – number of classes
- **learning_rate** (*float*) – learning rate
- **pretrained** (*Optional[str]*) – If true, will use the weights pretrained (using CPC) on Imagenet

`_CPCV2__compute_final_nb_c` (*patch_size*)

`_CPCV2__recover_z_shape` (*Z*, *b*)

```
static add_model_specific_args (parent_parser)
configure_optimizers ()
forward (img_1)
init_encoder ()
load_pretrained (encoder_name)
shared_step (batch)
training_step (batch, batch_nb)
validation_step (batch, batch_nb)
```

pl_bolts.models.self_supervised.cpc.networks module

```
class pl_bolts.models.self_supervised.cpc.networks.CPCResNet (sample_batch,
                                                             block,          layers,
                                                             zero_init_residual=False,
                                                             groups=1,
                                                             width_per_group=64,
                                                             re-
                                                             place_stride_with_dilation=None,
                                                             norm_layer=None)
```

Bases: `torch.nn.Module`

```
_make_layer (sample_batch, block, planes, blocks, stride=1, dilate=False, expansion=4)
```

```
flatten (x)
```

```
forward (x)
```

```
class pl_bolts.models.self_supervised.cpc.networks.LNBottleneck (sample_batch,
                                                                    inplanes,
                                                                    planes,
                                                                    stride=1,
                                                                    downsam-
                                                                    ple_conv=None,
                                                                    groups=1,
                                                                    base_width=64,
                                                                    dilation=1,
                                                                    norm_layer=None,
                                                                    expansion=4)
```

Bases: `torch.nn.Module`

```
_LNBottleneck__init_layer_norms (x, conv1, conv2, conv3, downsample_conv)
```

```
forward (x)
```

```
pl_bolts.models.self_supervised.cpc.networks.conv1x1 (in_planes,          out_planes,
                                                         stride=1)
    1x1 convolution
```

```
pl_bolts.models.self_supervised.cpc.networks.conv3x3 (in_planes,          out_planes,
                                                         stride=1,  groups=1,  dila-
                                                         tion=1)
    3x3 convolution with padding
```

```
pl_bolts.models.self_supervised.cpc.networks.cpc_resnet101 (sample_batch,
                                                              **kwargs)
```

```
pl_bolts.models.self_supervised.cpc.networks.cpc_resnet50(sample_batch,
                                                         **kwargs)
```

pl_bolts.models.self_supervised.cpc.transforms module

```
class pl_bolts.models.self_supervised.cpc.transforms.CPCEvalTransformsCIFAR10(patch_size=8,
                                                                              over-
                                                                              lap=4)
```

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=overlap)
```

Example:

```
# in a regular dataset
CIFAR10(..., transforms=CPCEvalTransformsCIFAR10())

# in a DataModule
module = CIFAR10DataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
↳ transforms=CPCEvalTransformsCIFAR10())
```

Parameters

- **patch_size** (`int`) – size of patches when cutting up the image into overlapping patches
- **overlap** (`int`) – how much to overlap patches

`__call__` (*inp*)

Call self as a function.

```
class pl_bolts.models.self_supervised.cpc.transforms.CPCEvalTransformsImageNet128(patch_size=,
                                                                              over-
                                                                              lap=16)
```

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=patch_size // 2)
```

Example:

```
# in a regular dataset
Imagenet(..., transforms=CPCEvalTransformsImageNet128())

# in a DataModule
```

(continues on next page)

(continued from previous page)

```
module = ImagenetDataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
↳ transforms=CPCEvalTransformsImageNet128())
```

Parameters

- **patch_size** (*int*) – size of patches when cutting up the image into overlapping patches
- **overlap** (*int*) – how much to overlap patches

__call__ (*inp*)

Call self as a function.

```
class pl_bolts.models.self_supervised.cpc.transforms.CPCEvalTransformsSTL10(patch_size=16,
over-
lap=8)
```

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=patch_size // 2)
```

Example:

```
# in a regular dataset
STL10(..., transforms=CPCEvalTransformsSTL10())

# in a DataModule
module = STL10DataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
↳ transforms=CPCEvalTransformsSTL10())
```

Parameters

- **patch_size** (*int*) – size of patches when cutting up the image into overlapping patches
- **overlap** (*int*) – how much to overlap patches

__call__ (*inp*)

Call self as a function.

```
class pl_bolts.models.self_supervised.cpc.transforms.CPCTrainTransformsCIFAR10(patch_size=8,
over-
lap=4)
```

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
img_jitter
col_jitter
```

(continues on next page)

(continued from previous page)

```

rnd_gray
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=patch_size // 2)

```

Example:

```

# in a regular dataset
CIFAR10(..., transforms=CPCTrainTransformsCIFAR10())

# in a DataModule
module = CIFAR10DataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
↳ transforms=CPCTrainTransformsCIFAR10())

```

Parameters

- **patch_size** – size of patches when cutting up the image into overlapping patches
- **overlap** – how much to overlap patches

`__call__`(*inp*)

Call self as a function.

class `pl_bolts.models.self_supervised.cpc.transforms.CPCTrainTransformsImageNet128` (*patch_size=32, overlap=16*)

Bases: `object`

Transforms used for CPC:

Transforms:

```

random_flip
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=patch_size // 2)

```

Example:

```

# in a regular dataset
Imagenet(..., transforms=CPCTrainTransformsImageNet128())

# in a DataModule
module = ImagenetDataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
↳ transforms=CPCTrainTransformsImageNet128())

```

Parameters

- **patch_size** (*int*) – size of patches when cutting up the image into overlapping patches
- **overlap** (*int*) – how much to overlap patches

`__call__`(*inp*)

Call self as a function.

```
class pl_bolts.models.self_supervised.cpc.transforms.CPCTrainTransformsSTL10(patch_size=16,
                                                                              overlap=8)
```

Bases: `object`

Transforms used for CPC:

Transforms:

```
random_flip
img_jitter
col_jitter
rnd_gray
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=patch_size // 2)
```

Example:

```
# in a regular dataset
STL10(..., transforms=CPCTrainTransformsSTL10())

# in a DataModule
module = STL10DataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,
                                       transforms=CPCTrainTransformsSTL10())
```

Parameters

- **patch_size** (`int`) – size of patches when cutting up the image into overlapping patches
- **overlap** (`int`) – how much to overlap patches

`__call__` (*inp*)

Call self as a function.

`pl_bolts.models.self_supervised.moco` package

Submodules

`pl_bolts.models.self_supervised.moco.callbacks` module

```
class pl_bolts.models.self_supervised.moco.callbacks.MocoLRScheduler(initial_lr=0.03,
                                                                       use_cosine_scheduler=False,
                                                                       schedule=(120,
                                                                       160),
                                                                       max_epochs=200)
```

Bases: `pytorch_lightning.Callback`

on_epoch_start (*trainer, pl_module*)

pl_bolts.models.self_supervised.moco.moco2_module module

Adapted from: <https://github.com/facebookresearch/moco>

Original work is: Copyright (c) Facebook, Inc. and its affiliates. All Rights Reserved This implementation is: Copyright (c) PyTorch Lightning, Inc. and its affiliates. All Rights Reserved

This implementation is licensed under Attribution-NonCommercial 4.0 International; You may not use this file except in compliance with the License.

You may obtain a copy of the License from the LICENSE file present in this folder.

```
class pl_bolts.models.self_supervised.moco.moco2_module.MocoV2 (base_encoder='resnet18',
                                                                emb_dim=128,
                                                                num_negatives=65536,
                                                                en-
                                                                coder_momentum=0.999,
                                                                soft-
                                                                max_temperature=0.07,
                                                                learn-
                                                                ing_rate=0.03,
                                                                momentum=0.9,
                                                                weight_decay=0.0001,
                                                                data_dir='./',
                                                                batch_size=256,
                                                                use_mlp=False,
                                                                num_workers=8,
                                                                *args,
                                                                **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of **Moco**

Paper authors: Xinlei Chen, Haoqi Fan, Ross Girshick, Kaiming He.

Code adapted from [facebookresearch/moco](https://github.com/facebookresearch/moco) to Lightning by:

- [William Falcon](#)

Example:: `from pl_bolts.models.self_supervised import MocoV2 model = MocoV2() trainer = Trainer() trainer.fit(model)`

CLI command:

```
# cifar10
python moco2_module.py --gpus 1

# imagenet
python moco2_module.py
    --gpus 8
    --dataset imagenet2012
    --data_dir /path/to/imagenet/
    --meta_dir /path/to/folder/with/meta.bin/
    --batch_size 32
```

Parameters

- **base_encoder** (`Union[str, Module]`) – torchvision model name or torch.nn.Module

- **emb_dim** (*int*) – feature dimension (default: 128)
- **num_negatives** (*int*) – queue size; number of negative keys (default: 65536)
- **encoder_momentum** (*float*) – moco momentum of updating key encoder (default: 0.999)
- **softmax_temperature** (*float*) – softmax temperature (default: 0.07)
- **learning_rate** (*float*) – the learning rate
- **momentum** (*float*) – optimizer momentum
- **weight_decay** (*float*) – optimizer weight decay
- **datamodule** – the DataModule (train, val, test dataloaders)
- **data_dir** (*str*) – the directory to store data
- **batch_size** (*int*) – batch size
- **use_mlp** (*bool*) – add an mlp to the encoders
- **num_workers** (*int*) – workers for the loaders

`_batch_shuffle_ddp` (*x*)

Batch shuffle, for making use of BatchNorm. * **Only support DistributedDataParallel (DDP) model.** *

`_batch_unshuffle_ddp` (*x, idx_unshuffle*)

Undo batch shuffle. * **Only support DistributedDataParallel (DDP) model.** *

`_dequeue_and_enqueue` (*keys*)

`_momentum_update_key_encoder` ()

Momentum update of the key encoder

`static add_model_specific_args` (*parent_parser*)

`configure_optimizers` ()

`forward` (*img_q, img_k*)

Input: *img_q*: a batch of query images *img_k*: a batch of key images

Output: logits, targets

`init_encoders` (*base_encoder*)

Override to add your own encoders

`training_step` (*batch, batch_idx*)

`validation_epoch_end` (*outputs*)

`validation_step` (*batch, batch_idx*)

`pl_bolts.models.self_supervised.moco.moco2_module.cli_main()`

`pl_bolts.models.self_supervised.moco.moco2_module.concat_all_gather` (*tensor*)

Performs all_gather operation on the provided tensors. * **Warning** *: `torch.distributed.all_gather` has no gradient.

pl_bolts.models.self_supervised.moco.transforms module

class pl_bolts.models.self_supervised.moco.transforms.**GaussianBlur** (*sigma=(0.1, 2.0)*)

Bases: `object`

Gaussian blur augmentation in SimCLR <https://arxiv.org/abs/2002.05709>

__call__ (*x*)

Call self as a function.

class pl_bolts.models.self_supervised.moco.transforms.**Moco2EvalCIFAR10Transforms** (*height=32*)

Bases: `object`

Moco 2 augmentation: <https://arxiv.org/pdf/2003.04297.pdf>

__call__ (*inp*)

Call self as a function.

class pl_bolts.models.self_supervised.moco.transforms.**Moco2EvalImagenetTransforms** (*height=128*)

Bases: `object`

Moco 2 augmentation: <https://arxiv.org/pdf/2003.04297.pdf>

__call__ (*inp*)

Call self as a function.

class pl_bolts.models.self_supervised.moco.transforms.**Moco2EvalSTL10Transforms** (*height=64*)

Bases: `object`

Moco 2 augmentation: <https://arxiv.org/pdf/2003.04297.pdf>

__call__ (*inp*)

Call self as a function.

class pl_bolts.models.self_supervised.moco.transforms.**Moco2TrainCIFAR10Transforms** (*height=32*)

Bases: `object`

Moco 2 augmentation: <https://arxiv.org/pdf/2003.04297.pdf>

__call__ (*inp*)

Call self as a function.

class pl_bolts.models.self_supervised.moco.transforms.**Moco2TrainImagenetTransforms** (*height=128*)

Bases: `object`

Moco 2 augmentation: <https://arxiv.org/pdf/2003.04297.pdf>

__call__ (*inp*)

Call self as a function.

class pl_bolts.models.self_supervised.moco.transforms.**Moco2TrainSTL10Transforms** (*height=64*)

Bases: `object`

Moco 2 augmentation: <https://arxiv.org/pdf/2003.04297.pdf>

__call__ (*inp*)

Call self as a function.

pl_bolts.models.self_supervised.simclr package

Submodules

pl_bolts.models.self_supervised.simclr.simclr_finetuner module

```
pl_bolts.models.self_supervised.simclr.simclr_finetuner.cli_main()
```

pl_bolts.models.self_supervised.simclr.simclr_module module

```
class pl_bolts.models.self_supervised.simclr.simclr_module.Projection(input_dim=2048,
                                                                    hid-
                                                                    den_dim=2048,
                                                                    out-
                                                                    put_dim=128)
```

Bases: `torch.nn.Module`

forward (*x*)

```
class pl_bolts.models.self_supervised.simclr.simclr_module.SimCLR(gpus,
                                                                    num_samples,
                                                                    batch_size,
                                                                    dataset,
                                                                    num_nodes=1,
                                                                    arch='resnet50',
                                                                    hid-
                                                                    den_mlp=2048,
                                                                    feat_dim=128,
                                                                    warmup_epochs=10,
                                                                    max_epochs=100,
                                                                    tempera-
                                                                    ture=0.1,
                                                                    first_conv=True,
                                                                    max-
                                                                    pool1=True,
                                                                    opti-
                                                                    mizer='adam',
                                                                    lars_wrapper=True,
                                                                    ex-
                                                                    clude_bn_bias=False,
                                                                    start_lr=0.0,
                                                                    learn-
                                                                    ing_rate=0.001,
                                                                    fi-
                                                                    nal_lr=0.0,
                                                                    weight_decay=1e-
                                                                    06,
                                                                    **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Parameters

- **batch_size** (`int`) – the batch size
- **num_samples** (`int`) – num samples in the dataset

- **warmup_epochs** (*int*) – epochs to warmup the lr for
- **lr** – the optimizer learning rate
- **opt_weight_decay** – the optimizer weight decay
- **loss_temperature** – the loss temperature

static add_model_specific_args (*parent_parser*)

configure_optimizers ()

exclude_from_wt_decay (*named_params, weight_decay, skip_list=['bias', 'bn']*)

forward (*x*)

init_model ()

nt_xent_loss (*out_1, out_2, temperature, eps=1e-06*)

assume out_1 and out_2 are normalized out_1: [batch_size, dim] out_2: [batch_size, dim]

optimizer_step (*epoch=None, batch_idx=None, optimizer=None, optimizer_idx=None, optimizer_closure=None, on_tpu=None, using_native_amp=None, using_lbfgs=None*)

Return type *None*

shared_step (*batch*)

training_step (*batch, batch_idx*)

validation_step (*batch, batch_idx*)

class `pl_bolts.models.self_supervised.simclr.simclr_module.SyncFunction` (**args, **kwargs*)

Bases: `torch.autograd.Function`

static backward (*ctx, grad_output*)

static forward (*ctx, tensor*)

`pl_bolts.models.self_supervised.simclr.simclr_module.cli_main` ()

`pl_bolts.models.self_supervised.simclr.transforms` module

class `pl_bolts.models.self_supervised.simclr.transforms.GaussianBlur` (*kernel_size, p=0.5, min=0.1, max=2.0*)

Bases: `object`

__call__ (*sample*)

Call self as a function.

class `pl_bolts.models.self_supervised.simclr.transforms.SimCLREvalDataTransform` (*input_height=2, gaussian_blur=True, jitter_strength=1, normalize=None*)

Bases: `pl_bolts.models.self_supervised.simclr.transforms.SimCLRTrainDataTransform`

Transforms for SimCLR

Transform:

```
Resize(input_height + 10, interpolation=3)
transforms.CenterCrop(input_height),
transforms.ToTensor()
```

Example:

```
from pl_bolts.models.self_supervised.simclr.transforms import _
↳ SimCLREvalDataTransform

transform = SimCLREvalDataTransform(input_height=32)
x = sample()
(xi, xj) = transform(x)
```

```
class pl_bolts.models.self_supervised.simclr.transforms.SimCLRFinetuneTransform(input_height=256,
                                                                                jitter_strength=1.0,
                                                                                normalize=None,
                                                                                eval_transform=None)
```

Bases: `object`

```
__call__(sample)
    Call self as a function.
```

```
class pl_bolts.models.self_supervised.simclr.transforms.SimCLRTrainDataTransform(input_height=256,
                                                                                gaussian_blur=True,
                                                                                jitter_strength=1.0,
                                                                                normalize=None)
```

Bases: `object`

Transforms for SimCLR

Transform:

```
RandomResizedCrop(size=self.input_height)
RandomHorizontalFlip()
RandomApply([color_jitter], p=0.8)
RandomGrayscale(p=0.2)
GaussianBlur(kernel_size=int(0.1 * self.input_height))
transforms.ToTensor()
```

Example:

```
from pl_bolts.models.self_supervised.simclr.transforms import _
↳ SimCLRTrainDataTransform

transform = SimCLRTrainDataTransform(input_height=32)
x = sample()
(xi, xj) = transform(x)
```

`__call__` (*sample*)
Call self as a function.

`pl_bolts.models.self_supervised.simsiam` package

Submodules

`pl_bolts.models.self_supervised.simsiam.models` module

class `pl_bolts.models.self_supervised.simsiam.models.MLP` (*input_dim=2048, hid-
den_size=4096, out-
put_dim=256*)

Bases: `torch.nn.Module`

forward (*x*)

Return type `Tensor`

class `pl_bolts.models.self_supervised.simsiam.models.SiameseArm` (*encoder=None, in-
put_dim=2048, hid-
den_size=4096, out-
put_dim=256*)

Bases: `torch.nn.Module`

forward (*x*)

Return type `Tuple[Tensor, Tensor, Tensor]`

pl_bolts.models.self_supervised.simsiam.simsiam_module module

```
class pl_bolts.models.self_supervised.simsiam.simsiam_module.SimSiam(gpus,
                                                                    num_samples,
                                                                    batch_size,
                                                                    dataset,
                                                                    num_nodes=1,
                                                                    arch='resnet50',
                                                                    hid-
                                                                    den_mlp=2048,
                                                                    feat_dim=128,
                                                                    warmup_epochs=10,
                                                                    max_epochs=100,
                                                                    tem-
                                                                    perature=0.1,
                                                                    first_conv=True,
                                                                    max-
                                                                    pool=True,
                                                                    opti-
                                                                    mizer='adam',
                                                                    lars_wrapper=True,
                                                                    ex-
                                                                    clude_bn_bias=False,
                                                                    start_lr=0.0,
                                                                    learn-
                                                                    ing_rate=0.001,
                                                                    fi-
                                                                    nal_lr=0.0,
                                                                    weight_decay=1e-
                                                                    06,
                                                                    **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

PyTorch Lightning implementation of [Exploring Simple Siamese Representation Learning \(SimSiam\)](#)

Paper authors: Xinlei Chen, Kaiming He.

Model implemented by:

- [Zvi Lapp](#)

Warning: Work in progress. This implementation is still being verified.

TODOs:

- verify on CIFAR-10
- verify on STL-10
- pre-train on imagenet

Example:

```

model = SimSiam()

dm = CIFAR10DataModule(num_workers=0)
dm.train_transforms = SimCLRTrainDataTransform(32)
dm.val_transforms = SimCLREvalDataTransform(32)

trainer = pl.Trainer()
trainer.fit(model, datamodule=dm)

```

Train:

```

trainer = Trainer()
trainer.fit(model)

```

CLI command:

```

# cifar10
python simsiam_module.py --gpus 1

# imagenet
python simsiam_module.py
    --gpus 8
    --dataset imagenet2012
    --data_dir /path/to/imagenet/
    --meta_dir /path/to/folder/with/meta.bin/
    --batch_size 32

```

Parameters

- **datamodule** – The datamodule
- **learning_rate** (*float*) – the learning rate
- **weight_decay** (*float*) – optimizer weight decay
- **input_height** – image input height
- **batch_size** (*int*) – the batch size
- **num_workers** – number of workers
- **warmup_epochs** (*int*) – num of epochs for scheduler warm up
- **max_epochs** (*int*) – max epochs for scheduler

static add_model_specific_args (*parent_parser*)

configure_optimizers ()

cosine_similarity (*a, b*)

exclude_from_wt_decay (*named_params, weight_decay, skip_list=['bias', 'bn']*)

forward (*x*)

init_model ()

optimizer_step (*epoch, batch_idx, optimizer, optimizer_idx, optimizer_closure=None, on_tpu=False, using_native_amp=False, using_lbfgs=False*)

Return type *None*

training_step (*batch, batch_idx*)

```
validation_step(batch, batch_idx)
```

```
pl_bolts.models.self_supervised.simsiam.simsiam_module.cli_main()
```

pl_bolts.models.self_supervised.swav package

```
class pl_bolts.models.self_supervised.swav.SwAV(gpus, num_samples, batch_size,
dataset, num_nodes=1,
arch='resnet50', hidden_mlp=2048,
feat_dim=128, warmup_epochs=10,
max_epochs=100,
nmb_prototypes=3000,
freeze_prototypes_epochs=1, tem-
perature=0.1, sinkhorn_iterations=3,
queue_length=0, queue_path='queue',
epoch_queue_starts=15,
crops_for_assign=[0, 1],
nmb_crops=[2, 6], first_conv=True,
maxpool1=True, optimizer='adam',
lars_wrapper=True, ex-
clude_bn_bias=False, start_lr=0.0,
learning_rate=0.001, final_lr=0.0,
weight_decay=1e-06, epsilon=0.05,
**kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Parameters

- **gpus** (`int`) – number of gpus per node used in training, passed to SwAV module to manage the queue and select distributed sinkhorn
- **num_nodes** (`int`) – number of nodes to train on
- **num_samples** (`int`) – number of image samples used for training
- **batch_size** (`int`) – batch size per GPU in ddp
- **dataset** (`str`) – dataset being used for train/val
- **arch** (`str`) – encoder architecture used for pre-training
- **hidden_mlp** (`int`) – hidden layer of non-linear projection head, set to 0 to use a linear projection head
- **feat_dim** (`int`) – output dim of the projection head
- **warmup_epochs** (`int`) – apply linear warmup for this many epochs
- **max_epochs** (`int`) – epoch count for pre-training
- **nmb_prototypes** (`int`) – count of prototype vectors
- **freeze_prototypes_epochs** (`int`) – epoch till which gradients of prototype layer are frozen
- **temperature** (`float`) – loss temperature
- **sinkhorn_iterations** (`int`) – iterations for sinkhorn normalization
- **queue_length** (`int`) – set queue when batch size is small, must be divisible by total batch-size (i.e. `total_gpus * batch_size`), set to 0 to remove the queue

- **queue_path** (*str*) – folder within the logs directory
- **epoch_queue_starts** (*int*) – start using the queue after this epoch
- **crops_for_assign** (*list*) – list of crop ids for computing assignment
- **nmb_crops** (*list*) – number of global and local crops, ex: [2, 6]
- **first_conv** (*bool*) – keep first conv same as the original resnet architecture, if set to false it is replaced by a kernel 3, stride 1 conv (cifar-10)
- **maxpool1** (*bool*) – keep first maxpool layer same as the original resnet architecture, if set to false, first maxpool is turned off (cifar10, maybe stl10)
- **optimizer** (*str*) – optimizer to use
- **lars_wrapper** (*bool*) – use LARS wrapper over the optimizer
- **exclude_bn_bias** (*bool*) – exclude batchnorm and bias layers from weight decay in optimizers
- **start_lr** (*float*) – starting lr for linear warmup
- **learning_rate** (*float*) – learning rate
- **final_lr** (*float*) – float = final learning rate for cosine weight decay
- **weight_decay** (*float*) – weight decay for optimizer
- **epsilon** (*float*) – epsilon val for swav assignments

static add_model_specific_args (*parent_parser*)

configure_optimizers ()

distributed_sinkhorn (*Q*, *nmb_iters*)

exclude_from_wt_decay (*named_params*, *weight_decay*, *skip_list*=['bias', 'bn'])

forward (*x*)

init_model ()

on_after_backward ()

on_train_epoch_end (*outputs*)

Return type None

on_train_epoch_start ()

optimizer_step (*epoch=None*, *batch_idx=None*, *optimizer=None*, *optimizer_idx=None*, *optimizer_closure=None*, *on_tpu=None*, *using_native_amp=None*, *using_lbfgs=None*)

Return type None

setup (*stage*)

shared_step (*batch*)

sinkhorn (*Q*, *nmb_iters*)

training_step (*batch*, *batch_idx*)

validation_step (*batch*, *batch_idx*)

```
class pl_bolts.models.self_supervised.swav.SwAVEvalDataTransform(normalize=None,
                                                                size_crops=[96,
                                                                36],
                                                                nmb_crops=[2,
                                                                4],
                                                                min_scale_crops=[0.33,
                                                                0.1],
                                                                max_scale_crops=[1,
                                                                0.33], gaussian_blur=True,
                                                                jitter_strength=1.0)

    Bases: pl_bolts.models.self_supervised.swav.transforms.SwAVTrainDataTransform

class pl_bolts.models.self_supervised.swav.SwAVFinetuneTransform(input_height=224,
                                                                jitter_strength=1.0,
                                                                normalize=None,
                                                                eval_transform=False)

    Bases: object

    __call__(sample)
        Call self as a function.

class pl_bolts.models.self_supervised.swav.SwAVTrainDataTransform(normalize=None,
                                                                size_crops=[96,
                                                                36],
                                                                nmb_crops=[2,
                                                                4],
                                                                min_scale_crops=[0.33,
                                                                0.1],
                                                                max_scale_crops=[1,
                                                                0.33], gaussian_blur=True,
                                                                jitter_strength=1.0)

    Bases: object

    __call__(sample)
        Call self as a function.

pl_bolts.models.self_supervised.swav.resnet18(**kwargs)
pl_bolts.models.self_supervised.swav.resnet50(**kwargs)
```

Submodules

pl_bolts.models.self_supervised.swav.swav_finetuner module

`pl_bolts.models.self_supervised.swav.swav_finetuner.cli_main()`

pl_bolts.models.self_supervised.swav.swav_module module

Adapted from official swav implementation: <https://github.com/facebookresearch/swav>

```
class pl_bolts.models.self_supervised.swav.swav_module.SwAV(gpus, num_samples,
    batch_size, dataset,
    num_nodes=1,
    arch='resnet50',
    hidden_mlp=2048,
    feat_dim=128,
    warmup_epochs=10,
    max_epochs=100,
    nmb_prototypes=3000,
    freeze_prototypes_epochs=1,
    temperature=0.1,
    sinkhorn_iterations=3,
    queue_length=0,
    queue_path='queue',
    epoch_queue_starts=15,
    crops_for_assign=[0,
1], nmb_crops=[2,
6], first_conv=True,
    maxpool1=True,
    optimizer='adam',
    lars_wrapper=True,
    ex-
clude_bn_bias=False,
    start_lr=0.0, learn-
ing_rate=0.001,
    final_lr=0.0,
    weight_decay=1e-
06, epsilon=0.05,
    **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Parameters

- **gpus** (`int`) – number of gpus per node used in training, passed to SwAV module to manage the queue and select distributed sinkhorn
- **num_nodes** (`int`) – number of nodes to train on
- **num_samples** (`int`) – number of image samples used for training
- **batch_size** (`int`) – batch size per GPU in ddp
- **dataset** (`str`) – dataset being used for train/val
- **arch** (`str`) – encoder architecture used for pre-training

- **hidden_mlp** (*int*) – hidden layer of non-linear projection head, set to 0 to use a linear projection head
- **feat_dim** (*int*) – output dim of the projection head
- **warmup_epochs** (*int*) – apply linear warmup for this many epochs
- **max_epochs** (*int*) – epoch count for pre-training
- **nmb_prototypes** (*int*) – count of prototype vectors
- **freeze_prototypes_epochs** (*int*) – epoch till which gradients of prototype layer are frozen
- **temperature** (*float*) – loss temperature
- **sinkhorn_iterations** (*int*) – iterations for sinkhorn normalization
- **queue_length** (*int*) – set queue when batch size is small, must be divisible by total batch-size (i.e. `total_gpus * batch_size`), set to 0 to remove the queue
- **queue_path** (*str*) – folder within the logs directory
- **epoch_queue_starts** (*int*) – start using the queue after this epoch
- **crops_for_assign** (*list*) – list of crop ids for computing assignment
- **nmb_crops** (*list*) – number of global and local crops, ex: [2, 6]
- **first_conv** (*bool*) – keep first conv same as the original resnet architecture, if set to false it is replace by a kernel 3, stride 1 conv (cifar-10)
- **maxpool1** (*bool*) – keep first maxpool layer same as the original resnet architecture, if set to false, first maxpool is turned off (cifar10, maybe stl10)
- **optimizer** (*str*) – optimizer to use
- **lars_wrapper** (*bool*) – use LARS wrapper over the optimizer
- **exclude_bn_bias** (*bool*) – exclude batchnorm and bias layers from weight decay in optimizers
- **start_lr** (*float*) – starting lr for linear warmup
- **learning_rate** (*float*) – learning rate
- **final_lr** (*float*) – float = final learning rate for cosine weight decay
- **weight_decay** (*float*) – weight decay for optimizer
- **epsilon** (*float*) – epsilon val for swav assignments

```
static add_model_specific_args (parent_parser)
configure_optimizers ()
distributed_sinkhorn (Q, nmb_iters)
exclude_from_wt_decay (named_params, weight_decay, skip_list=['bias', 'bn'])
forward (x)
init_model ()
on_after_backward ()
on_train_epoch_end (outputs)
```

Return type None

```

on_train_epoch_start ()

optimizer_step (epoch=None, batch_idx=None, optimizer=None, optimizer_idx=None,
                 optimizer_closure=None, on_tpu=None, using_native_amp=None, us-
                 ing_lbfgs=None)

    Return type None

setup (stage)

shared_step (batch)

sinkhorn (Q, nmb_iters)

training_step (batch, batch_idx)

validation_step (batch, batch_idx)

pl_bolts.models.self_supervised.swav.swav_module.cli_main()

```

pl_bolts.models.self_supervised.swav.swav_resnet module

Adapted from: <https://github.com/facebookresearch/swav/blob/master/src/resnet50.py>

```

class pl_bolts.models.self_supervised.swav.swav_resnet.BasicBlock (inplanes,
                                                                    planes,
                                                                    stride=1,
                                                                    downsam-
                                                                    ple=None,
                                                                    groups=1,
                                                                    base_width=64,
                                                                    dilation=1,
                                                                    norm_layer=None)

    Bases: torch.nn.Module

    forward (x)

    expansion = 1

class pl_bolts.models.self_supervised.swav.swav_resnet.Bottleneck (inplanes,
                                                                    planes,
                                                                    stride=1,
                                                                    downsam-
                                                                    ple=None,
                                                                    groups=1,
                                                                    base_width=64,
                                                                    dilation=1,
                                                                    norm_layer=None)

    Bases: torch.nn.Module

    forward (x)

    expansion = 4

class pl_bolts.models.self_supervised.swav.swav_resnet.MultiPrototypes (output_dim,
                                                                    nmb_prototypes)

    Bases: torch.nn.Module

    forward (x)

```

```
class pl_bolts.models.self_supervised.swav.swav_resnet.ResNet (block,      layers,
                                                         zero_init_residual=False,
                                                         groups=1,
                                                         widen=1,
                                                         width_per_group=64,
                                                         re-
                                                         place_stride_with_dilation=None,
                                                         norm_layer=None,
                                                         normalize=False,
                                                         output_dim=0,
                                                         hidden_mlp=0,
                                                         nmb_prototypes=0,
                                                         eval_mode=False,
                                                         first_conv=True,
                                                         maxpool1=True)

Bases: torch.nn.Module

_make_layer (block, planes, blocks, stride=1, dilate=False)

forward (inputs)

forward_backbone (x)

forward_head (x)

pl_bolts.models.self_supervised.swav.swav_resnet.conv1x1 (in_planes,      out_planes,
                                                         stride=1)
    1x1 convolution

pl_bolts.models.self_supervised.swav.swav_resnet.conv3x3 (in_planes,      out_planes,
                                                         stride=1,      groups=1,
                                                         dilation=1)
    3x3 convolution with padding

pl_bolts.models.self_supervised.swav.swav_resnet.resnet18 (**kwargs)

pl_bolts.models.self_supervised.swav.swav_resnet.resnet50 (**kwargs)

pl_bolts.models.self_supervised.swav.swav_resnet.resnet50w2 (**kwargs)

pl_bolts.models.self_supervised.swav.swav_resnet.resnet50w4 (**kwargs)

pl_bolts.models.self_supervised.swav.swav_resnet.resnet50w5 (**kwargs)
```

pl_bolts.models.self_supervised.swav.transforms module

```
class pl_bolts.models.self_supervised.swav.transforms.GaussianBlur (kernel_size,
                                                         p=0.5,
                                                         min=0.1,
                                                         max=2.0)

Bases: object

__call__ (sample)
    Call self as a function.
```

```

class pl_bolts.models.self_supervised.swav.transforms.SwAVEvalDataTransform (normalize=None,
                                                                    size_crops=[96,
                                                                    36],
                                                                    nmb_crops=[2,
                                                                    4],
                                                                    min_scale_crops=[0.
                                                                    0.1],
                                                                    max_scale_crops=[1,
                                                                    0.33],
                                                                    gaus-
                                                                    sian_blur=True,
                                                                    jit-
                                                                    ter_strength=1.0)

    Bases: pl_bolts.models.self_supervised.swav.transforms.SwAVTrainDataTransform

class pl_bolts.models.self_supervised.swav.transforms.SwAVFinetuneTransform (input_height=224,
                                                                    jit-
                                                                    ter_strength=1.0,
                                                                    nor-
                                                                    mal-
                                                                    ize=None,
                                                                    eval_transform=False)

    Bases: object

    __call__ (sample)
        Call self as a function.

class pl_bolts.models.self_supervised.swav.transforms.SwAVTrainDataTransform (normalize=None,
                                                                    size_crops=[96,
                                                                    36],
                                                                    nmb_crops=[2,
                                                                    4],
                                                                    min_scale_crops=[0.
                                                                    0.1],
                                                                    max_scale_crops=[
                                                                    0.33],
                                                                    gaus-
                                                                    sian_blur=True,
                                                                    jit-
                                                                    ter_strength=1.0)

    Bases: object

    __call__ (sample)
        Call self as a function.

```

Submodules

`pl_bolts.models.self_supervised.evaluator` module

```

class pl_bolts.models.self_supervised.evaluator.Flatten
    Bases: torch.nn.Module

    forward (input_tensor)

```

```
class pl_bolts.models.self_supervised.evaluator.SSLEvaluator (n_input, n_classes,  
                                                         n_hidden=512,  
                                                         p=0.1)  
  
    Bases: torch.nn.Module  
  
    forward (x)
```

`pl_bolts.models.self_supervised.resnets` module

```
class pl_bolts.models.self_supervised.resnets.ResNet (block,                                layers,  
                                                         num_classes=1000,  
                                                         zero_init_residual=False,  
                                                         groups=1,  
                                                         width_per_group=64,      re-  
                                                         place_stride_with_dilation=None,  
                                                         norm_layer=None,      re-  
                                                         turn_all_feature_maps=False,  
                                                         first_conv=True,      max-  
                                                         pool1=True)  
  
    Bases: torch.nn.Module  
  
    _make_layer (block, planes, blocks, stride=1, dilate=False)  
  
    forward (x)
```

```
pl_bolts.models.self_supervised.resnets.resnet18 (pretrained=False, progress=True,  
                                                         **kwargs)
```

ResNet-18 model from “[Deep Residual Learning for Image Recognition](#)”

Parameters

- **pretrained** (`bool`) – If True, returns a model pre-trained on ImageNet
- **progress** (`bool`) – If True, displays a progress bar of the download to stderr

```
pl_bolts.models.self_supervised.resnets.resnet34 (pretrained=False, progress=True,  
                                                         **kwargs)
```

ResNet-34 model from “[Deep Residual Learning for Image Recognition](#)”

Parameters

- **pretrained** – If True, returns a model pre-trained on ImageNet
- **progress** – If True, displays a progress bar of the download to stderr

```
pl_bolts.models.self_supervised.resnets.resnet50 (pretrained=False, progress=True,  
                                                         **kwargs)
```

ResNet-50 model from “[Deep Residual Learning for Image Recognition](#)”

Parameters

- **pretrained** (`bool`) – If True, returns a model pre-trained on ImageNet
- **progress** (`bool`) – If True, displays a progress bar of the download to stderr

```
pl_bolts.models.self_supervised.resnets.resnet101 (pretrained=False, progress=True,  
                                                         **kwargs)
```

ResNet-101 model from “[Deep Residual Learning for Image Recognition](#)”

Parameters

- **pretrained** (`bool`) – If True, returns a model pre-trained on ImageNet
- **progress** (`bool`) – If True, displays a progress bar of the download to stderr


```
pl_bolts.models.self_supervised.resnets.resnet152(pretrained=False, progress=True,
                                                    **kwargs)
```

ResNet-152 model from “[Deep Residual Learning for Image Recognition](#)”

Parameters

- **pretrained** (`bool`) – If True, returns a model pre-trained on ImageNet
- **progress** (`bool`) – If True, displays a progress bar of the download to stderr

```
pl_bolts.models.self_supervised.resnets.resnext50_32x4d(pretrained=False,
                                                         progress=True, **kwargs)
```

ResNeXt-50 32x4d model from “[Aggregated Residual Transformation for Deep Neural Networks](#)”

Parameters

- **pretrained** (`bool`) – If True, returns a model pre-trained on ImageNet
- **progress** (`bool`) – If True, displays a progress bar of the download to stderr

```
pl_bolts.models.self_supervised.resnets.resnext101_32x8d(pretrained=False,
                                                           progress=True,
                                                           **kwargs)
```

ResNeXt-101 32x8d model from “[Aggregated Residual Transformation for Deep Neural Networks](#)”

Parameters

- **pretrained** (`bool`) – If True, returns a model pre-trained on ImageNet
- **progress** (`bool`) – If True, displays a progress bar of the download to stderr

```
pl_bolts.models.self_supervised.resnets.wide_resnet50_2(pretrained=False,
                                                          progress=True, **kwargs)
```

Wide ResNet-50-2 model from “[Wide Residual Networks](#)” The model is the same as ResNet except for the bottleneck number of channels which is twice larger in every block. The number of channels in outer 1x1 convolutions is the same, e.g. last block in ResNet-50 has 2048-512-2048 channels, and in Wide ResNet-50-2 has 2048-1024-2048.

Parameters

- **pretrained** (`bool`) – If True, returns a model pre-trained on ImageNet
- **progress** (`bool`) – If True, displays a progress bar of the download to stderr

```
pl_bolts.models.self_supervised.resnets.wide_resnet101_2(pretrained=False,
                                                           progress=True,
                                                           **kwargs)
```

Wide ResNet-101-2 model from “[Wide Residual Networks](#)” The model is the same as ResNet except for the bottleneck number of channels which is twice larger in every block. The number of channels in outer 1x1 convolutions is the same, e.g. last block in ResNet-50 has 2048-512-2048 channels, and in Wide ResNet-50-2 has 2048-1024-2048.

Parameters

- **pretrained** (`bool`) – If True, returns a model pre-trained on ImageNet
- **progress** (`bool`) – If True, displays a progress bar of the download to stderr

pl_bolts.models.self_supervised.ssl_finetuner module

```
class pl_bolts.models.self_supervised.ssl_finetuner.SSLFineTuner (backbone,
                                                                in_features=2048,
                                                                num_classes=1000,
                                                                epochs=100,
                                                                hid-
                                                                den_dim=None,
                                                                dropout=0.0,
                                                                learn-
                                                                ing_rate=0.1,
                                                                weight_decay=1e-
                                                                06,          nes-
                                                                terov=False,
                                                                sched-
                                                                uler_type='cosine',
                                                                de-
                                                                cay_epochs=[60,
                                                                80],
                                                                gamma=0.1,
                                                                final_lr=0.0)
```

Bases: `pytorch_lightning.LightningModule`

Finetunes a self-supervised learning backbone using the standard evaluation protocol of a singler layer MLP with 1024 units

Example:

```
from pl_bolts.utils.self_supervised import SSLFineTuner
from pl_bolts.models.self_supervised import CPCV2
from pl_bolts.datamodules import CIFAR10DataModule
from pl_bolts.models.self_supervised.cpc.transforms import _
↳CPCEvalTransformsCIFAR10,
↳CPCTrainTransformsCIFAR10

# pretrained model
backbone = CPCV2.load_from_checkpoint(PATH, strict=False)

# dataset + transforms
dm = CIFAR10DataModule(data_dir='.')
dm.train_transforms = CPCTrainTransformsCIFAR10()
dm.val_transforms = CPCEvalTransformsCIFAR10()

# finetuner
finetuner = SSLFineTuner(backbone, in_features=backbone.z_dim, num_
↳classes=backbone.num_classes)

# train
trainer = pl.Trainer()
trainer.fit(finetuner, dm)

# test
trainer.test(datamodule=dm)
```

Parameters

- **backbone** (`Module`) – a pretrained model
- **in_features** (`int`) – feature dim of backbone outputs
- **num_classes** (`int`) – classes of the dataset
- **hidden_dim** (`Optional[int]`) – dim of the MLP (1024 default used in self-supervised literature)

configure_optimizers ()

on_train_epoch_start ()

Return type `None`

shared_step (*batch*)

test_step (*batch, batch_idx*)

training_step (*batch, batch_idx*)

validation_step (*batch, batch_idx*)

pl_bolts.models.vision package

class `pl_bolts.models.vision.PixelCNN` (*input_channels*, *hidden_channels=256*,
num_blocks=5)

Bases: `torch.nn.Module`

Implementation of `Pixel CNN`.

Paper authors: Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, Koray Kavukcuoglu

Implemented by:

- William Falcon

Example:

```
>>> from pl_bolts.models.vision import PixelCNN
>>> import torch
...
>>> model = PixelCNN(input_channels=3)
>>> x = torch.rand(5, 3, 64, 64)
>>> out = model(x)
...
>>> out.shape
torch.Size([5, 3, 64, 64])
```

conv_block (*input_channels*)

forward (*z*)

class `pl_bolts.models.vision.SemSegment` (*lr=0.01*, *num_classes=19*, *num_layers=5*, *fea-*
tures_start=64, bilinear=False)

Bases: `pytorch_lightning.LightningModule`

Basic model for semantic segmentation. Uses UNet architecture by default.

The default parameters in this model are for the KITTI dataset. Note, if you'd like to use this model as is, you will first need to download the KITTI dataset yourself. You can download the dataset [here](#).

Implemented by:

- [Annika Brundyn](#)

Parameters

- **num_layers** (*int*) – number of layers in each side of U-net (default 5)
- **features_start** (*int*) – number of features in first layer (default 64)
- **bilinear** (*bool*) – whether to use bilinear interpolation (True) or transposed convolutions (default) for upsampling.
- **lr** (*float*) – learning (default 0.01)

static `add_model_specific_args` (*parent_parser*)

configure_optimizers ()

forward (*x*)

training_step (*batch*, *batch_nb*)

validation_epoch_end (*outputs*)

validation_step (*batch*, *batch_idx*)

class `pl_bolts.models.vision.UNet` (*num_classes*, *input_channels=3*, *num_layers=5*, *features_start=64*, *bilinear=False*)

Bases: `torch.nn.Module`

Paper: [U-Net: Convolutional Networks for Biomedical Image Segmentation](#)

Paper authors: Olaf Ronneberger, Philipp Fischer, Thomas Brox

Implemented by:

- [Annika Brundyn](#)
- [Akshay Kulkarni](#)

Parameters

- **num_classes** (*int*) – Number of output classes required
- **input_channels** (*int*) – Number of channels in input images (default 3)
- **num_layers** (*int*) – Number of layers in each side of U-net (default 5)
- **features_start** (*int*) – Number of features in first layer (default 64)
- **bilinear** (*bool*) – Whether to use bilinear interpolation or transposed convolutions (default) for upsampling.

forward (*x*)

Subpackages

pl_bolts.models.vision.image_gpt package

Submodules

pl_bolts.models.vision.image_gpt.gpt2 module

class pl_bolts.models.vision.image_gpt.gpt2.**Block** (*embed_dim, heads*)
 Bases: `torch.nn.Module`

forward (*x*)

class pl_bolts.models.vision.image_gpt.gpt2.**GPT2** (*embed_dim, heads, layers, num_positions, vocab_size, num_classes*)
 Bases: `pytorch_lightning.LightningModule`

GPT-2 from [language Models are Unsupervised Multitask Learners](#)

Paper by: Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever

Implementation contributed by:

- [Teddy Koker](#)

Example:

```
from pl_bolts.models import GPT2

seq_len = 17
batch_size = 32
vocab_size = 16
x = torch.randint(0, vocab_size, (seq_len, batch_size))
model = GPT2(embed_dim=32, heads=2, layers=2, num_positions=seq_len, vocab_
↪size=vocab_size, num_classes=4)
results = model(x)
```

_init_embeddings ()

_init_layers ()

_init_sos_token ()

forward (*x, classify=False*)

Expect input as shape [sequence len, batch] If classify, return classification logits

pl_bolts.models.vision.image_gpt.igpt_module module

```
class pl_bolts.models.vision.image_gpt.igpt_module.ImageGPT(embed_dim=16,
                                                            heads=2,      lay-
                                                            ers=2,      pixels=28,
                                                            vocab_size=16,
                                                            num_classes=10,
                                                            classify=False,
                                                            batch_size=64,
                                                            learning_rate=0.01,
                                                            steps=25000,
                                                            data_dir='.',
                                                            num_workers=8,
                                                            **kwargs)
```

Bases: `pytorch_lightning.LightningModule`

Paper: [Generative Pretraining from Pixels](#) [original paper code].

Paper by: Mark Che, Alec Radford, Rewon Child, Jeff Wu, Heewoo Jun, Prafulla Dhariwal, David Luan, Ilya Sutskever

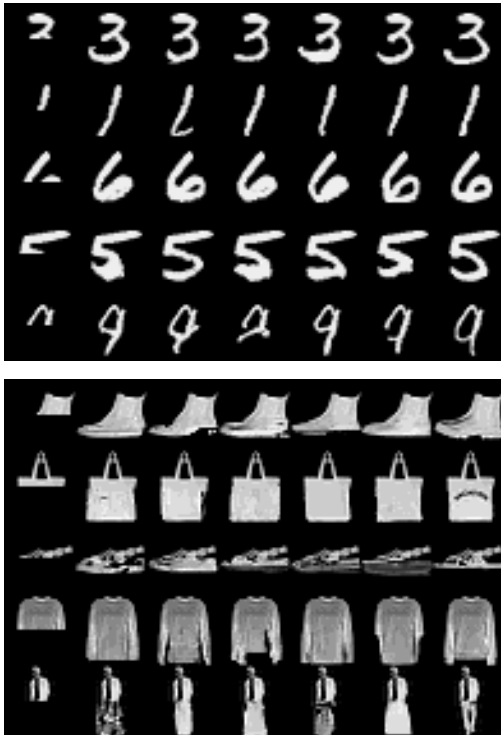
Implementation contributed by:

- [Teddy Koker](#)

Original repo with results and more implementation details:

- <https://github.com/teddykoker/image-gpt>

Example Results (Photo credits: Teddy Koker):



Default arguments:

Table 2: Argument Defaults

Argument	Default	iGPT-S (Chen et al.)
<code>-embed_dim</code>	16	512
<code>-heads</code>	2	8
<code>-layers</code>	8	24
<code>-pixels</code>	28	32
<code>-vocab_size</code>	16	512
<code>-num_classes</code>	10	10
<code>-batch_size</code>	64	128
<code>-learning_rate</code>	0.01	0.01
<code>-steps</code>	25000	1000000

Example:

```
import pytorch_lightning as pl
from pl_bolts.models.vision import ImageGPT

dm = MNISTDataModule('.')
model = ImageGPT(dm)

pl.Trainer(gpu=4).fit(model)
```

As script:

```
cd pl_bolts/models/vision/image_gpt
python igpt_module.py --learning_rate 1e-2 --batch_size 32 --gpus 4
```

Parameters

- **embed_dim** (`int`) – the embedding dim
- **heads** (`int`) – number of attention heads
- **layers** (`int`) – number of layers
- **pixels** (`int`) – number of input pixels
- **vocab_size** (`int`) – vocab size
- **num_classes** (`int`) – number of classes in the input
- **classify** (`bool`) – true if should classify
- **batch_size** (`int`) – the batch size
- **learning_rate** (`float`) – learning rate
- **steps** (`int`) – number of steps for cosine annealing
- **data_dir** (`str`) – where to store data
- **num_workers** (`int`) – num_data workers

```
static add_model_specific_args (parent_parser)
configure_optimizers ()
forward (x, classify=False)
test_epoch_end (outs)
```

```
test_step (batch, batch_idx)
training_step (batch, batch_idx)
validation_epoch_end (outs)
validation_step (batch, batch_idx)

pl_bolts.models.vision.image_gpt.igpt_module._shape_input (x)
    shape batch of images for input into GPT2 model

pl_bolts.models.vision.image_gpt.igpt_module.cli_main()
```

Submodules

pl_bolts.models.vision.pixel_cnn module

PixelCNN Implemented by: William Falcon Reference: <https://arxiv.org/pdf/1905.09272.pdf> (page 15) Accessed: May 14, 2020

```
class pl_bolts.models.vision.pixel_cnn.PixelCNN (input_channels,          hid-
                                                den_channels=256, num_blocks=5)
```

Bases: `torch.nn.Module`

Implementation of [Pixel CNN](#).

Paper authors: Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, Koray Kavukcuoglu

Implemented by:

- William Falcon

Example:

```
>>> from pl_bolts.models.vision import PixelCNN
>>> import torch
...
>>> model = PixelCNN(input_channels=3)
>>> x = torch.rand(5, 3, 64, 64)
>>> out = model(x)
...
>>> out.shape
torch.Size([5, 3, 64, 64])
```

```
conv_block (input_channels)
```

```
forward (z)
```

pl_bolts.models.vision.segmentation module

```
class pl_bolts.models.vision.segmentation.SemSegment (lr=0.01,      num_classes=19,
                                                         num_layers=5,          fea-
                                                         tures_start=64,      bilin-
                                                         ear=False)
```

Bases: `pytorch_lightning.LightningModule`

Basic model for semantic segmentation. Uses UNet architecture by default.

The default parameters in this model are for the KITTI dataset. Note, if you'd like to use this model as is, you will first need to download the KITTI dataset yourself. You can download the dataset [here](#).

Implemented by:

- [Annika Brundyn](#)

Parameters

- **num_layers** (*int*) – number of layers in each side of U-net (default 5)
- **features_start** (*int*) – number of features in first layer (default 64)
- **bilinear** (*bool*) – whether to use bilinear interpolation (True) or transposed convolutions (default) for upsampling.
- **lr** (*float*) – learning (default 0.01)

```
static add_model_specific_args (parent_parser)
```

```
configure_optimizers ()
```

```
forward (x)
```

```
training_step (batch, batch_nb)
```

```
validation_epoch_end (outputs)
```

```
validation_step (batch, batch_idx)
```

```
pl_bolts.models.vision.segmentation.cli_main ()
```

pl_bolts.models.vision.unet module

```
class pl_bolts.models.vision.unet.DoubleConv (in_ch, out_ch)
```

Bases: `torch.nn.Module`

[Conv2d => BatchNorm (optional) => ReLU] x 2

```
forward (x)
```

```
class pl_bolts.models.vision.unet.Down (in_ch, out_ch)
```

Bases: `torch.nn.Module`

Downscale with MaxPool => DoubleConvolution block

```
forward (x)
```

```
class pl_bolts.models.vision.unet.UNet (num_classes, input_channels=3, num_layers=5, features_start=64, bilinear=False)
```

Bases: `torch.nn.Module`

Paper: [U-Net: Convolutional Networks for Biomedical Image Segmentation](#)

Paper authors: Olaf Ronneberger, Philipp Fischer, Thomas Brox

Implemented by:

- [Annika Brundyn](#)
- [Akshay Kulkarni](#)

Parameters

- **num_classes** (*int*) – Number of output classes required

- **input_channels** (*int*) – Number of channels in input images (default 3)
- **num_layers** (*int*) – Number of layers in each side of U-net (default 5)
- **features_start** (*int*) – Number of features in first layer (default 64)
- **bilinear** (*bool*) – Whether to use bilinear interpolation or transposed convolutions (default) for upsampling.

forward (*x*)

class `pl_bolts.models.vision.unet.Up` (*in_ch, out_ch, bilinear=False*)

Bases: `torch.nn.Module`

Upsampling (by either bilinear interpolation or transpose convolutions) followed by concatenation of feature map from contracting path, followed by DoubleConv.

forward (*x1, x2*)

32.5.2 Submodules

`pl_bolts.models.mnist_module` module

class `pl_bolts.models.mnist_module.LitMNIST` (*hidden_dim=128, learning_rate=0.001, batch_size=32, num_workers=4, data_dir="", **kwargs*)

Bases: `pytorch_lightning.LightningModule`

static `add_model_specific_args` (*parent_parser*)

configure_optimizers ()

forward (*x*)

prepare_data ()

test_dataloader ()

test_step (*batch, batch_idx*)

train_dataloader ()

training_step (*batch, batch_idx*)

val_dataloader ()

validation_step (*batch, batch_idx*)

`pl_bolts.models.mnist_module.cli_main` ()

32.6 `pl_bolts.losses` package

32.6.1 Submodules

`pl_bolts.losses.object_detection` module

Loss functions for Object Detection task

`pl_bolts.losses.object_detection.giou_loss(preds, target)`

Calculates the generalized intersection over union loss.

It has been proposed in [Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression](#).

Parameters

- **preds** (`Tensor`) – an Nx4 batch of prediction bounding boxes with representation `[x_min, y_min, x_max, y_max]`
- **target** (`Tensor`) – an Mx4 batch of target bounding boxes with representation `[x_min, y_min, x_max, y_max]`

Example

```
>>> import torch
>>> from pl_bolts.losses.object_detection import giou_loss
>>> preds = torch.tensor([[100, 100, 200, 200]])
>>> target = torch.tensor([[150, 150, 250, 250]])
>>> giou_loss(preds, target)
tensor([[1.0794]])
```

Return type `Tensor`

Returns GIoU loss in an NxM tensor containing the pairwise GIoU loss for every element in `preds` and `target`, where N is the number of prediction bounding boxes and M is the number of target bounding boxes

`pl_bolts.losses.object_detection.iou_loss(preds, target)`

Calculates the intersection over union loss.

Parameters

- **preds** (`Tensor`) – batch of prediction bounding boxes with representation `[x_min, y_min, x_max, y_max]`
- **target** (`Tensor`) – batch of target bounding boxes with representation `[x_min, y_min, x_max, y_max]`

Example

```
>>> import torch
>>> from pl_bolts.losses.object_detection import iou_loss
>>> preds = torch.tensor([[100, 100, 200, 200]])
>>> target = torch.tensor([[150, 150, 250, 250]])
>>> iou_loss(preds, target)
tensor([[0.8571]])
```

Return type `Tensor`

Returns IoU loss

pl_bolts.losses.rl module

Loss functions for the RL models

`pl_bolts.losses.rl.double_dqn_loss(batch, net, target_net, gamma=0.99)`

Calculates the mse loss using a mini batch from the replay buffer. This uses an improvement to the original DQN loss by using the double dqn. This is shown by using the actions of the train network to pick the value from the target network. This code is heavily commented in order to explain the process clearly

Parameters

- **batch** (`Tuple[Tensor, Tensor]`) – current mini batch of replay data
- **net** (`Module`) – main training network
- **target_net** (`Module`) – target network of the main training network
- **gamma** (`float`) – discount factor

Return type `Tensor`

Returns loss

`pl_bolts.losses.rl.dqn_loss(batch, net, target_net, gamma=0.99)`

Calculates the mse loss using a mini batch from the replay buffer

Parameters

- **batch** (`Tuple[Tensor, Tensor]`) – current mini batch of replay data
- **net** (`Module`) – main training network
- **target_net** (`Module`) – target network of the main training network
- **gamma** (`float`) – discount factor

Return type `Tensor`

Returns loss

`pl_bolts.losses.rl.per_dqn_loss(batch, batch_weights, net, target_net, gamma=0.99)`

Calculates the mse loss with the priority weights of the batch from the PER buffer

Parameters

- **batch** (`Tuple[Tensor, Tensor]`) – current mini batch of replay data
- **batch_weights** (`List`) – how each of these samples are weighted in terms of priority
- **net** (`Module`) – main training network
- **target_net** (`Module`) – target network of the main training network
- **gamma** (`float`) – discount factor

Return type `Tuple[Tensor, ndarray]`

Returns loss and batch_weights

pl_bolts.losses.self_supervised_learning module

class pl_bolts.losses.self_supervised_learning.**AmdimNCELoss** (*tclip*)

Bases: `torch.nn.Module`

Compute the NCE scores for predicting `r_src->r_trg`.

forward (*anchor_representations, positive_representations, mask_mat*)

Parameters

- **anchor_representations** – (batch_size, emb_dim)
- **positive_representations** – (emb_dim, n_batch * w* h) (ie: nb_feat_vectors x embedding_dim)
- **mask_mat** – (n_batch_gpu, n_batch)

Output: raw_scores: (n_batch_gpu, n_locs) nce_scores: (n_batch_gpu, n_locs) lgt_reg : scalar

class pl_bolts.losses.self_supervised_learning.**CPCTask** (*num_input_channels,*
target_dim=64, *em-*
bed_scale=0.1)

Bases: `torch.nn.Module`

Loss used in CPC

compute_loss_h (*targets, preds, i*)

forward (*Z*)

class pl_bolts.losses.self_supervised_learning.**FeatureMapContrastiveTask** (*comparisons='00,*
11',
tclip=10.0,
bidi-
rec-
tional=True)

Bases: `torch.nn.Module`

Performs an anchor, positive negative pair comparison for each each tuple of feature maps passed.

```
# extract feature maps
pos_0, pos_1, pos_2 = encoder(x_pos)
anc_0, anc_1, anc_2 = encoder(x_anchor)

# compare only the 0th feature maps
task = FeatureMapContrastiveTask('00')
loss, regularizer = task((pos_0), (anc_0))

# compare (pos_0 to anc_1) and (pos_0, anc_2)
task = FeatureMapContrastiveTask('01, 02')
losses, regularizer = task((pos_0, pos_1, pos_2), (anc_0, anc_1, anc_2))
loss = losses.sum()

# compare (pos_1 vs a anc_random)
task = FeatureMapContrastiveTask('0r')
loss, regularizer = task((pos_0, pos_1, pos_2), (anc_0, anc_1, anc_2))
```

```
# with bidirectional the comparisons are done both ways
task = FeatureMapContrastiveTask('01, 02')
```

(continues on next page)

(continued from previous page)

```
# will compare the following:
# 01: (pos_0, anc_1), (anc_0, pos_1)
# 02: (pos_0, anc_2), (anc_0, pos_2)
```

Parameters

- **comparisons** (`str`) – groupings of feature map indices to compare (zero indexed, ‘r’ means random) ex: ‘00, 1r’
- **tclip** (`float`) – stability clipping value
- **bidirectional** (`bool`) – if true, does the comparison both ways

__FeatureMapContrastiveTask__cache_dimension_masks (*args)

__FeatureMapContrastiveTask__compare_maps (m1, m2)

__sample_src_ftr (r_cnv, masks)

feat_size_w_mask (w, feature_map)

forward (anchor_maps, positive_maps)

Takes in a set of tuples, each tuple has two feature maps with all matching dimensions

Example

```
>>> import torch
>>> from pytorch_lightning import seed_everything
>>> seed_everything(0)
0
>>> a1 = torch.rand(3, 5, 2, 2)
>>> a2 = torch.rand(3, 5, 2, 2)
>>> b1 = torch.rand(3, 5, 2, 2)
>>> b2 = torch.rand(3, 5, 2, 2)
...
>>> task = FeatureMapContrastiveTask('01, 11')
...
>>> losses, regularizer = task((a1, a2), (b1, b2))
>>> losses
tensor([2.2351, 2.1902])
>>> regularizer
tensor(0.0324)
```

static parse_map_indexes (comparisons)

Example:

```
>>> FeatureMapContrastiveTask.parse_map_indexes('11')
[(1, 1)]
>>> FeatureMapContrastiveTask.parse_map_indexes('11,59')
[(1, 1), (5, 9)]
>>> FeatureMapContrastiveTask.parse_map_indexes('11,59, 2r')
[(1, 1), (5, 9), (2, -1)]
```

`pl_bolts.losses.self_supervised_learning.nt_xent_loss` (out_1, out_2, temperature)

Loss used in SimCLR

`pl_bolts.losses.self_supervised_learning.tanh_clip` (x, clip_val=10.0)

soft clip values to the range [-clip_val, +clip_val]

32.7 pl_bolts.optimizers package

class `pl_bolts.optimizers.LARSWrapper` (*optimizer, eta=0.02, clip=True, eps=1e-08*)

Bases: `object`

Wrapper that adds LARS scheduling to any optimizer. This helps stability with huge batch sizes.

Parameters

- **optimizer** – torch optimizer
- **eta** – LARS coefficient (trust)
- **clip** – True to clip LR
- **eps** – adaptive_lr stability coefficient

step (*closure=None*)

update_p (*p, group, weight_decay*)

property defaults

property param_groups

property state

class `pl_bolts.optimizers.LinearWarmupCosineAnnealingLR` (*optimizer, warmup_epochs, max_epochs, warmup_start_lr=0.0, eta_min=0.0, last_epoch=-1*)

Bases: `torch.optim.lr_scheduler._LRScheduler`

Sets the learning rate of each parameter group to follow a linear warmup schedule between `warmup_start_lr` and `base_lr` followed by a cosine annealing schedule between `base_lr` and `eta_min`.

Warning: It is recommended to call `step()` for `LinearWarmupCosineAnnealingLR` after each iteration as calling it after each epoch will keep the starting lr at `warmup_start_lr` for the first epoch which is 0 in most cases.

Warning: passing `epoch` to `step()` is being deprecated and comes with an `EPOCH_DEPRECATION_WARNING`. It calls the `_get_closed_form_lr()` method for this scheduler instead of `get_lr()`. Though this does not change the behavior of the scheduler, when passing `epoch` param to `step()`, the user should call the `step()` function before calling `train` and `validation` methods.

Example

```

>>> layer = nn.Linear(10, 1)
>>> optimizer = Adam(layer.parameters(), lr=0.02)
>>> scheduler = LinearWarmupCosineAnnealingLR(optimizer, warmup_epochs=10, max_
    ↪ epochs=40)
>>> #
>>> # the default case
>>> for epoch in range(40):
...     # train(...)
...     # validate(...)
...     scheduler.step()
>>> #
>>> # passing epoch param case
>>> for epoch in range(40):
...     scheduler.step(epoch)
...     # train(...)
...     # validate(...)

```

Parameters

- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **warmup_epochs** (*int*) – Maximum number of iterations for linear warmup
- **max_epochs** (*int*) – Maximum number of iterations
- **warmup_start_lr** (*float*) – Learning rate to start the linear warmup. Default: 0.
- **eta_min** (*float*) – Minimum learning rate. Default: 0.
- **last_epoch** (*int*) – The index of last epoch. Default: -1.

_get_closed_form_lr()

Called when epoch is passed as a param to the *step* function of the scheduler.

Return type `List[float]`

get_lr()

Compute learning rate using chainable form of the scheduler

Return type `List[float]`

32.7.1 Submodules

pl_bolts.optimizers.lars_scheduling module

References

- <https://github.com/NVIDIA/apex/blob/master/apex/parallel/LARC.py>
- <https://arxiv.org/pdf/1708.03888.pdf>
- <https://github.com/noahgolmant/pytorch-lars/blob/master/lars.py>

```

class pl_bolts.optimizers.lars_scheduling.LARSWrapper(optimizer, eta=0.02,
    clip=True, eps=1e-08)

```

Bases: `object`

Wrapper that adds LARS scheduling to any optimizer. This helps stability with huge batch sizes.

Parameters

- **optimizer** – torch optimizer
- **eta** – LARS coefficient (trust)
- **clip** – True to clip LR
- **eps** – adaptive_lr stability coefficient

step (*closure=None*)

update_p (*p, group, weight_decay*)

property defaults

property param_groups

property state

pl_bolts.optimizers.lr_scheduler module

```
class pl_bolts.optimizers.lr_scheduler.LinearWarmupCosineAnnealingLR(optimizer,
                                                                    warmup_epochs,
                                                                    max_epochs,
                                                                    warmup_start_lr=0.0,
                                                                    eta_min=0.0,
                                                                    last_epoch=-1)
```

Bases: torch.optim.lr_scheduler._LRScheduler

Sets the learning rate of each parameter group to follow a linear warmup schedule between warmup_start_lr and base_lr followed by a cosine annealing schedule between base_lr and eta_min.

Warning: It is recommended to call `step()` for `LinearWarmupCosineAnnealingLR` after each iteration as calling it after each epoch will keep the starting lr at warmup_start_lr for the first epoch which is 0 in most cases.

Warning: passing epoch to `step()` is being deprecated and comes with an EPOCH_DEPRECATION_WARNING. It calls the `_get_closed_form_lr()` method for this scheduler instead of `get_lr()`. Though this does not change the behavior of the scheduler, when passing epoch param to `step()`, the user should call the `step()` function before calling train and validation methods.

Example

```
>>> layer = nn.Linear(10, 1)
>>> optimizer = Adam(layer.parameters(), lr=0.02)
>>> scheduler = LinearWarmupCosineAnnealingLR(optimizer, warmup_epochs=10, max_
↳ epochs=40)
>>> #
>>> # the default case
>>> for epoch in range(40):
...     # train(...)
```

(continues on next page)

(continued from previous page)

```

...     # validate(...)
...     scheduler.step()
>>> #
>>> # passing epoch param case
>>> for epoch in range(40):
...     scheduler.step(epoch)
...     # train(...)
...     # validate(...)

```

Parameters

- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **warmup_epochs** (*int*) – Maximum number of iterations for linear warmup
- **max_epochs** (*int*) – Maximum number of iterations
- **warmup_start_lr** (*float*) – Learning rate to start the linear warmup. Default: 0.
- **eta_min** (*float*) – Minimum learning rate. Default: 0.
- **last_epoch** (*int*) – The index of last epoch. Default: -1.

_get_closed_form_lr()

Called when epoch is passed as a param to the *step* function of the scheduler.

Return type `List[float]`

get_lr()

Compute learning rate using chainable form of the scheduler

Return type `List[float]`

32.8 pl_bolts.transforms package

32.8.1 Subpackages

pl_bolts.transforms.self_supervised package

Submodules

pl_bolts.transforms.self_supervised.ssl_transforms module

```

class pl_bolts.transforms.self_supervised.ssl_transforms.Patchify(patch_size,
                                                                    over-
                                                                    lap_size)

```

Bases: `object`

__call__(*x*)

Call self as a function.

```

class pl_bolts.transforms.self_supervised.ssl_transforms.RandomTranslateWithReflect(max_tran

```

Bases: `object`

Translate image randomly Translate vertically and horizontally by n pixels where n is integer drawn uniformly independently for each axis from $[-\text{max_translation}, \text{max_translation}]$. Fill the uncovered blank area with reflect padding.

```
__call__(old_image)
    Call self as a function.
```

32.8.2 Submodules

`pl_bolts.transforms.dataset_normalizations` module

```
pl_bolts.transforms.dataset_normalizations.cifar10_normalization()
pl_bolts.transforms.dataset_normalizations.imagenet_normalization()
pl_bolts.transforms.dataset_normalizations.stl10_normalization()
```


PYTHON MODULE INDEX

p

`pl_bolts.callbacks`, 177
`pl_bolts.callbacks.byol_updates`, 189
`pl_bolts.callbacks.data_monitor`, 190
`pl_bolts.callbacks.knn_online`, 193
`pl_bolts.callbacks.printing`, 193
`pl_bolts.callbacks.ssl_online`, 195
`pl_bolts.callbacks.variational`, 196
`pl_bolts.callbacks.verifications`, 183
`pl_bolts.callbacks.verifications.base`, 183
`pl_bolts.callbacks.verifications.batch_gradient`, 185
`pl_bolts.callbacks.vision`, 187
`pl_bolts.callbacks.vision.confused_logit`, 187
`pl_bolts.callbacks.vision.image_generation`, 188
`pl_bolts.datamodules`, 196
`pl_bolts.datamodules.async_data_loader`, 217
`pl_bolts.datamodules.binary_mnist_datamodule`, 218
`pl_bolts.datamodules.cifar10_datamodule`, 219
`pl_bolts.datamodules.cityscapes_datamodule`, 222
`pl_bolts.datamodules.experience_source`, 224
`pl_bolts.datamodules.fashion_mnist_datamodule`, 227
`pl_bolts.datamodules.imagenet_datamodule`, 229
`pl_bolts.datamodules.kitti_datamodule`, 231
`pl_bolts.datamodules.mnist_datamodule`, 233
`pl_bolts.datamodules.sklearn_datamodule`, 234
`pl_bolts.datamodules.ssl_imagenet_datamodule`, 237
`pl_bolts.datamodules.stl10_datamodule`, 237
`pl_bolts.datamodules.vision_datamodule`, 240
`pl_bolts.datamodules.voc_detection_datamodule`, 241
`pl_bolts.datasets`, 242
`pl_bolts.datasets.base_dataset`, 249
`pl_bolts.datasets.cifar10_dataset`, 249
`pl_bolts.datasets.concat_dataset`, 251
`pl_bolts.datasets.dummy_dataset`, 252
`pl_bolts.datasets.imagenet_dataset`, 254
`pl_bolts.datasets.kitti_dataset`, 255
`pl_bolts.datasets.mnist_dataset`, 255
`pl_bolts.datasets.ssl_and_dim_datasets`, 256
`pl_bolts.losses`, 382
`pl_bolts.losses.object_detection`, 382
`pl_bolts.losses.rl`, 384
`pl_bolts.losses.self_supervised_learning`, 385
`pl_bolts.metrics`, 256
`pl_bolts.metrics.aggregation`, 256
`pl_bolts.metrics.object_detection`, 256
`pl_bolts.models`, 257
`pl_bolts.models.autoencoders`, 265
`pl_bolts.models.autoencoders.basic_ae`, 267
`pl_bolts.models.autoencoders.basic_ae.basic_ae_module`, 267
`pl_bolts.models.autoencoders.basic_vae`, 268
`pl_bolts.models.autoencoders.basic_vae.basic_vae_module`, 269
`pl_bolts.models.autoencoders.components`, 270
`pl_bolts.models.detection`, 272
`pl_bolts.models.detection.components`, 273
`pl_bolts.models.detection.components._supported_modules`, 273
`pl_bolts.models.detection.components.torchvision_backend`, 273

[pl_bolts.models.detection.faster_rcnn](#), [343](#)
[274](#) [pl_bolts.models.self_supervised.byol.models](#),
[pl_bolts.models.detection.faster_rcnn.backbones](#), [344](#)
[275](#) [pl_bolts.models.self_supervised.cpc](#), [345](#)
[pl_bolts.models.detection.faster_rcnn.faster_rcnn_model](#), [345](#)
[276](#) [pl_bolts.models.self_supervised.cpc.cpc_finetuner](#),
[349](#)
[pl_bolts.models.gans](#), [277](#) [pl_bolts.models.self_supervised.cpc.cpc_module](#),
[pl_bolts.models.gans.basic](#), [279](#) [349](#)
[pl_bolts.models.gans.basic.basic_gan_model](#), [349](#)
[279](#) [pl_bolts.models.self_supervised.cpc.networks](#),
[350](#)
[pl_bolts.models.gans.basic.components](#), [pl_bolts.models.self_supervised.cpc.transforms](#),
[281](#) [351](#)
[pl_bolts.models.gans.dcgan](#), [281](#) [pl_bolts.models.self_supervised.evaluator](#),
[pl_bolts.models.gans.dcgan.components](#), [371](#)
[281](#) [pl_bolts.models.self_supervised.moco](#),
[pl_bolts.models.gans.dcgan.dcgan_module](#), [354](#)
[282](#) [pl_bolts.models.self_supervised.moco.callbacks](#),
[354](#)
[pl_bolts.models.mnist_module](#), [382](#) [pl_bolts.models.self_supervised.moco.moco2_module](#),
[pl_bolts.models.regression](#), [283](#) [355](#)
[pl_bolts.models.regression.linear_regression](#), [355](#)
[285](#) [pl_bolts.models.self_supervised.moco.transforms](#),
[pl_bolts.models.regression.logistic_regression](#), [357](#)
[286](#) [pl_bolts.models.self_supervised.resnets](#),
[372](#)
[pl_bolts.models.rl](#), [286](#) [pl_bolts.models.self_supervised.simclr](#),
[pl_bolts.models.rl.common](#), [299](#) [358](#)
[pl_bolts.models.rl.common.agents](#), [299](#) [pl_bolts.models.self_supervised.simclr.simclr_finetuner](#),
[pl_bolts.models.rl.common.cli](#), [301](#) [358](#)
[pl_bolts.models.rl.common.gym_wrappers](#), [301](#) [pl_bolts.models.self_supervised.simclr.simclr_module](#),
[358](#)
[pl_bolts.models.rl.common.memory](#), [302](#) [pl_bolts.models.self_supervised.simclr.transforms](#),
[pl_bolts.models.rl.common.networks](#), [305](#) [359](#)
[pl_bolts.models.rl.double_dqn_model](#), [308](#) [pl_bolts.models.self_supervised.simsiam](#),
[pl_bolts.models.rl.dqn_model](#), [310](#) [361](#)
[pl_bolts.models.rl.dueling_dqn_model](#), [313](#) [pl_bolts.models.self_supervised.simsiam.models](#),
[361](#)
[pl_bolts.models.rl.noisy_dqn_model](#), [314](#) [pl_bolts.models.self_supervised.simsiam.simsiam_module](#),
[pl_bolts.models.rl.per_dqn_model](#), [316](#) [362](#)
[pl_bolts.models.rl.reinforce_model](#), [318](#) [pl_bolts.models.self_supervised.ssl_finetuner](#),
[pl_bolts.models.rl.vanilla_policy_gradient_model](#), [320](#) [374](#)
[pl_bolts.models.self_supervised](#), [322](#) [pl_bolts.models.self_supervised.swav](#),
[pl_bolts.models.self_supervised.amdim](#), [333](#) [364](#)
[333](#) [pl_bolts.models.self_supervised.swav.swav_finetuner](#),
[pl_bolts.models.self_supervised.amdim.amdim_module](#), [367](#)
[337](#) [pl_bolts.models.self_supervised.swav.swav_module](#),
[pl_bolts.models.self_supervised.amdim.datasets](#), [367](#) [pl_bolts.models.self_supervised.swav.swav_resnet](#),
[339](#) [pl_bolts.models.self_supervised.swav.transforms](#),
[pl_bolts.models.self_supervised.amdim.networks](#), [369](#) [370](#)
[339](#) [pl_bolts.models.vision](#), [375](#)
[pl_bolts.models.self_supervised.amdim.transforms](#), [370](#) [pl_bolts.models.vision.image_gpt](#), [377](#)
[340](#) [pl_bolts.models.vision.image_gpt.gpt2](#),
[pl_bolts.models.self_supervised.byol](#), [343](#) [377](#)
[343](#) [pl_bolts.models.self_supervised.byol.byol_module](#), [377](#)

`pl_bolts.models.vision.image_gpt.igpt_module,`
378
`pl_bolts.models.vision.pixel_cnn,` 380
`pl_bolts.models.vision.segmentation,` 380
`pl_bolts.models.vision.unet,` 381
`pl_bolts.optimizers,` 387
`pl_bolts.optimizers.lars_scheduling,` 388
`pl_bolts.optimizers.lr_scheduler,` 389
`pl_bolts.transforms,` 390
`pl_bolts.transforms.dataset_normalizations,`
391
`pl_bolts.transforms.self_supervised,` 390
`pl_bolts.transforms.self_supervised.ssl_transforms,`
390

Symbols

`_CPCV2__compute_final_nb_c()`
 (*pl_bolts.models.self_supervised.CPCV2*
method), 326
`_CPCV2__compute_final_nb_c()`
 (*pl_bolts.models.self_supervised.cpc.CPCV2*
method), 345
`_CPCV2__compute_final_nb_c()`
 (*pl_bolts.models.self_supervised.cpc.cpc_module.CPCV2*
method), 349
`_CPCV2__recover_z_shape()`
 (*pl_bolts.models.self_supervised.CPCV2*
method), 326
`_CPCV2__recover_z_shape()`
 (*pl_bolts.models.self_supervised.cpc.CPCV2*
method), 345
`_CPCV2__recover_z_shape()`
 (*pl_bolts.models.self_supervised.cpc.cpc_module.CPCV2*
method), 349
`_ConfusedLogitCallback__draw_sample()`
 (*pl_bolts.callbacks.ConfusedLogitCallback*
static method), 182
`_ConfusedLogitCallback__draw_sample()`
 (*pl_bolts.callbacks.vision.confused_logit.ConfusedLogitCallback*
static method), 188
`_FeatureMapContrastiveTask__cache_dimension_masks()`
 (*pl_bolts.losses.self_supervised_learning.FeatureMapContrastiveTask*
method), 386
`_FeatureMapContrastiveTask__compare_maps()`
 (*pl_bolts.losses.self_supervised_learning.FeatureMapContrastiveTask*
method), 386
`_LNBottleneck__init_layer_norms()`
 (*pl_bolts.models.self_supervised.cpc.networks.LNBottleneck*
method), 350
`__call__()` (*pl_bolts.datamodules.vocdetetection_datamodule.Compose*
method), 241
`__call__()` (*pl_bolts.models.rl.common.agents.Agent*
method), 299
`__call__()` (*pl_bolts.models.rl.common.agents.PolicyAgent*
method), 300
`__call__()` (*pl_bolts.models.rl.common.agents.ValueAgent*
method), 300
`__call__()` (*pl_bolts.models.self_supervised.amdim.AMDIMEvalTransf*
method), 335
`__call__()` (*pl_bolts.models.self_supervised.amdim.AMDIMEvalTransf*
method), 335
`__call__()` (*pl_bolts.models.self_supervised.amdim.AMDIMEvalTransf*
method), 335
`__call__()` (*pl_bolts.models.self_supervised.amdim.AMDIMTrainTransf*
method), 336
`__call__()` (*pl_bolts.models.self_supervised.amdim.AMDIMTrainTransf*
method), 336
`__call__()` (*pl_bolts.models.self_supervised.amdim.AMDIMTrainTransf*
method), 337
`__call__()` (*pl_bolts.models.self_supervised.amdim.transforms.AMDIM*
method), 340
`__call__()` (*pl_bolts.models.self_supervised.amdim.transforms.AMDIM*
method), 341
`__call__()` (*pl_bolts.models.self_supervised.amdim.transforms.AMDIM*
method), 341
`__call__()` (*pl_bolts.models.self_supervised.amdim.transforms.AMDIM*
method), 342
`__call__()` (*pl_bolts.models.self_supervised.amdim.transforms.AMDIM*
method), 342
`__call__()` (*pl_bolts.models.self_supervised.amdim.transforms.AMDIM*
method), 342
`__call__()` (*pl_bolts.models.self_supervised.cpc.CPCEvalTransformsC*
method), 346
`__call__()` (*pl_bolts.models.self_supervised.cpc.CPCEvalTransformsIm*
method), 346
`__call__()` (*pl_bolts.models.self_supervised.cpc.CPCEvalTransformsST*
method), 347
`__call__()` (*pl_bolts.models.self_supervised.cpc.CPCTrainTransformsC*
method), 347
`__call__()` (*pl_bolts.models.self_supervised.cpc.CPCTrainTransformsIm*
method), 348
`__call__()` (*pl_bolts.models.self_supervised.cpc.CPCTrainTransformsS*
method), 349
`__call__()` (*pl_bolts.models.self_supervised.cpc.transforms.CPCEvalTr*
method), 351
`__call__()` (*pl_bolts.models.self_supervised.cpc.transforms.CPCEvalTr*
method), 352
`__call__()` (*pl_bolts.models.self_supervised.cpc.transforms.CPCEvalTr*
method), 352

398 Index

`_default_transforms()` (`pl_bolts.datamodules.CityscapesDataModule` method), 202
`_default_transforms()` (`pl_bolts.datamodules.KittiDataModule` method), 209
`_default_transforms()` (`pl_bolts.datamodules.SSLLImagenetDataModule` method), 213
`_default_transforms()` (`pl_bolts.datamodules.STL10DataModule` method), 214
`_default_transforms()` (`pl_bolts.datamodules.VOCDetectionDataModule` method), 216
`_default_transforms()` (`pl_bolts.datamodules.cityscapes_datamodule.CityscapesDataModule` method), 224
`_default_transforms()` (`pl_bolts.datamodules.kitti_datamodule.KittiDataModule` method), 232
`_default_transforms()` (`pl_bolts.datamodules.ssl_imagenet_datamodule.SSLLImagenetDataModule` method), 237
`_default_transforms()` (`pl_bolts.datamodules.stl10_datamodule.STL10DataModule` method), 239
`_default_transforms()` (`pl_bolts.datamodules.vocdetection_datamodule.VOCDetectionDataModule` method), 242
`_dequeue_and_enqueue()` (`pl_bolts.models.self_supervised.MocoV2` method), 327
`_dequeue_and_enqueue()` (`pl_bolts.models.self_supervised.moco.moco2_module.MocoV2` method), 356
`_disc_step()` (`pl_bolts.models.gans.DCGAN` method), 278
`_disc_step()` (`pl_bolts.models.gans.dcgan.dcgan_module.DCGAN` method), 282
`_download_from_url()` (`pl_bolts.datasets.LightDataset` method), 242
`_download_from_url()` (`pl_bolts.datasets.base_dataset.LightDataset` method), 249
`_evaluate_iou()` (in `module pl_bolts.models.detection.faster_rcnn.faster_rcnn_module`), 277
`_extract_archive_save_torch()` (`pl_bolts.datasets.CIFAR10` method), 244
`_extract_archive_save_torch()` (`pl_bolts.datasets.cifar10_dataset.CIFAR10` method), 250
`_field_defaults` (`pl_bolts.datamodules.experience_source.Experience` attribute), 225
`_field_defaults` (`pl_bolts.models.rl.common.memory.Experience` attribute), 303
`_fields` (`pl_bolts.datamodules.experience_source.Experience` attribute), 225
`_fields` (`pl_bolts.models.rl.common.memory.Experience` attribute), 303
`_fields_defaults` (`pl_bolts.datamodules.experience_source.Experience` attribute), 225
`_fields_defaults` (`pl_bolts.models.rl.common.memory.Experience` attribute), 303
`_forward_acts()` (`pl_bolts.models.self_supervised.amdim.AMDIMEnc` method), 334
`_forward_acts()` (`pl_bolts.models.self_supervised.amdim.networks.AL` method), 339
`_gen_step()` (`pl_bolts.models.gans.DCGAN` method), 278
`_gen_step()` (`pl_bolts.models.gans.dcgan.dcgan_module.DCGAN` method), 282
`_get_closed_form_lr()` (`pl_bolts.optimizers.LinearWarmupCosineAnnealingLR` method), 390
`_get_closed_form_lr()` (`pl_bolts.optimizers.lr_scheduler.LinearWarmupCosineAnnealing` method), 390
`_get_conv_out()` (`pl_bolts.models.rl.common.networks.CNN` method), 305
`_get_conv_out()` (`pl_bolts.models.rl.common.networks.DuelingCNN` method), 306
`_get_conv_out()` (`pl_bolts.models.rl.common.networks.NoisyCNN` method), 307
`_get_disc_loss()` (`pl_bolts.models.gans.DCGAN` method), 278
`_get_disc_loss()` (`pl_bolts.models.gans.dcgan.dcgan_module.DCGAN` method), 282
`_get_discriminator()` (`pl_bolts.models.gans.DCGAN` method), 279
`_get_discriminator()` (`pl_bolts.models.gans.dcgan.dcgan_module.DCGAN` method), 282
`_get_fake_pred()` (`pl_bolts.models.gans.DCGAN` method), 279
`_get_fake_pred()` (`pl_bolts.models.gans.dcgan.dcgan_module.DCGAN` method), 282
`_get_gen_loss()` (`pl_bolts.models.gans.DCGAN` method), 279
`_get_gen_loss()` (`pl_bolts.models.gans.dcgan.dcgan_module.DCGAN` method), 282
`_get_generator()` (`pl_bolts.models.gans.DCGAN` method), 279
`_get_generator()` (`pl_bolts.models.gans.dcgan.dcgan_module.DCGAN` method), 282

<code>_get_input_array_copy()</code> (<code>pl_bolts.callbacks.verification.base.VerificationBase</code> method), 183	(<code>pl_bolts.models.gans.dcgan.components.DCGANGenerator</code> static method), 281
<code>_get_noise()</code> (<code>pl_bolts.models.gans.DCGAN</code> method), 279	<code>_make_layer()</code> (<code>pl_bolts.models.autoencoders.components.ResNetDeco</code> method), 271
<code>_get_noise()</code> (<code>pl_bolts.models.gans.dcgan.dcgan_module.DCGANModule</code> method), 283	<code>_make_layer()</code> (<code>pl_bolts.models.autoencoders.components.ResNetEnc</code> method), 271
<code>_get_splits()</code> (<code>pl_bolts.datamodules.vision_datamodule.VisionDataModule</code> method), 240	<code>_make_layer()</code> (<code>pl_bolts.models.self_supervised.cpc.networks.CPCRes</code> method), 250
<code>_get_submodule_names()</code> (<code>pl_bolts.callbacks.ModuleDataMonitor</code> method), 179	<code>_make_layer()</code> (<code>pl_bolts.models.self_supervised.resnets.ResNet</code> method), 372
<code>_get_submodule_names()</code> (<code>pl_bolts.callbacks.data_monitor.ModuleDataMonitor</code> method), 191	<code>_make_layer()</code> (<code>pl_bolts.models.self_supervised.swav.swav_resnet.Res</code> method), 370
<code>_init_datasets()</code> (<code>pl_bolts.datamodules.SklearnDataModule</code> method), 211	<code>_model_forward()</code> (<code>pl_bolts.callbacks.verification.base.VerificationBa</code> method), 184
<code>_init_datasets()</code> (<code>pl_bolts.datamodules.sklearn_datamodule.SklearnDataModule</code> method), 235	<code>_momentum_update_key_encoder()</code> (<code>pl_bolts.models.self_supervised.MocoV2</code> method), 327
<code>_init_embeddings()</code> (<code>pl_bolts.models.GPT2</code> method), 262	<code>_momentum_update_key_encoder()</code> (<code>pl_bolts.models.self_supervised.moco.moco2_module.MocoV2</code> method), 356
<code>_init_embeddings()</code> (<code>pl_bolts.models.vision.image_gpt.gpt2.GPT2</code> method), 377	<code>_plot()</code> (<code>pl_bolts.callbacks.ConfusedLogitCallback</code> method), 182
<code>_init_layers()</code> (<code>pl_bolts.models.GPT2</code> method), 262	<code>_plot()</code> (<code>pl_bolts.callbacks.vision.confused_logit.ConfusedLogitCallback</code> method), 188
<code>_init_layers()</code> (<code>pl_bolts.models.vision.image_gpt.gpt2.GPT2</code> method), 377	<code>_prepare_subset()</code> (<code>pl_bolts.datasets.LightDataset</code> static method), 242
<code>_init_sos_token()</code> (<code>pl_bolts.models.GPT2</code> method), 262	<code>_prepare_subset()</code> (<code>pl_bolts.datasets.base_dataset.LightDataset</code> static method), 249
<code>_init_sos_token()</code> (<code>pl_bolts.models.vision.image_gpt.gpt2.GPT2</code> method), 377	<code>_prepare_voc_instance()</code> (in module <code>pl_bolts.datamodules.vocdetection_datamodule</code>), 242
<code>_is_gzip()</code> (in module <code>pl_bolts.datasets.imagenet_dataset</code>), 254	<code>_raise()</code> (<code>pl_bolts.callbacks.verification.base.VerificationCallbackBase</code> method), 184
<code>_is_logger_available()</code> (<code>pl_bolts.callbacks.data_monitor.DataMonitorBase</code> method), 190	<code>_random_bbox()</code> (<code>pl_bolts.datasets.DummyDetectionDataset</code> method), 246
<code>_is_tar()</code> (in module <code>pl_bolts.datasets.imagenet_dataset</code>), 254	<code>_random_bbox()</code> (<code>pl_bolts.datasets.dummy_dataset.DummyDetectionD</code> method), 252
<code>_is_targz()</code> (in module <code>pl_bolts.datasets.imagenet_dataset</code>), 254	<code>_register_hook()</code> (<code>pl_bolts.callbacks.ModuleDataMonitor</code> method), 179
<code>_is_tarxz()</code> (in module <code>pl_bolts.datasets.imagenet_dataset</code>), 254	<code>_register_hook()</code> (<code>pl_bolts.callbacks.data_monitor.ModuleDataMon</code> method), 191
<code>_is_zip()</code> (in module <code>pl_bolts.datasets.imagenet_dataset</code>), 254	<code>_replace()</code> (<code>pl_bolts.datamodules.experience_source.Experience</code> method), 225
<code>_make()</code> (<code>pl_bolts.datamodules.experience_source.Experience</code> class method), 225	<code>_replace()</code> (<code>pl_bolts.models.rl.common.memory.Experience</code> method), 303
<code>_make()</code> (<code>pl_bolts.models.rl.common.memory.Experience</code> class method), 303	<code>_scan_step()</code> (<code>pl_bolts.models.VAE</code> method), 259
<code>_make_disc_block()</code> (<code>pl_bolts.models.gans.dcgan.components.DCGANDiscriminator</code> static method), 281	<code>_run_step()</code> (<code>pl_bolts.models.autoencoders.VAE</code> method), 266
<code>_make_gen_block()</code>	<code>_run_step()</code> (<code>pl_bolts.models.autoencoders.basic_vae.basic_vae_modu</code> method), 269
	<code>_discriminator_c_ftr()</code> (<code>pl_bolts.losses.self_supervised_learning.FeatureMapContrastive</code> method), 386

`_shape_input()` (in module `pl_bolts.models.vision.image_gpt.igpt_module`), 380
`_split_dataset()` (`pl_bolts.datamodules.vision_datamodule.VisionDataModule` static method), 240
`_unpickle()` (`pl_bolts.datasets.CIFAR10` static method), 244
`_unpickle()` (`pl_bolts.datasets.cifar10_dataset.CIFAR10Dataset` static method), 250
`_verify_archive()` (in module `pl_bolts.datasets.imagenet_dataset`), 254
`_verify_splits()` (`pl_bolts.datamodules.ImagenetDataModule` static method), 207
`_verify_splits()` (`pl_bolts.datamodules.SSLImagenetDataModule` static method), 213
`_verify_splits()` (`pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule` static method), 230
`_verify_splits()` (`pl_bolts.datamodules.ssl_imagenet_datamodule.SSLImagenetDataModule` static method), 237
`_weights_init()` (`pl_bolts.models.gans.DCGAN` static method), 279
`_weights_init()` (`pl_bolts.models.gans.dcgan.dcgan_module.DCGANModule` static method), 283
A
`accuracy()` (in module `pl_bolts.metrics`), 256
`accuracy()` (in module `pl_bolts.metrics.aggregation`), 256
`action()` (`pl_bolts.datamodules.experience_source.ExperienceSource` property), 225
`action()` (`pl_bolts.models.rl.common.memory.ExperienceMemory` property), 303
`add()` (`pl_bolts.models.rl.common.memory.MeanBuffer` static method), 303
`add_base_args()` (in module `pl_bolts.models.rl.common.cli`), 301
`add_model_specific_args()` (`pl_bolts.models.AE` static method), 258
`add_model_specific_args()` (`pl_bolts.models.autoencoders.AE` static method), 265
`add_model_specific_args()` (`pl_bolts.models.autoencoders.basic_ae.basic_ae_module.BasicAEModule` static method), 268
`add_model_specific_args()` (`pl_bolts.models.autoencoders.basic_vae.basic_vae_module.BasicVAEModule` static method), 269
`add_model_specific_args()` (`pl_bolts.models.autoencoders.VAE` static method), 266
`add_model_specific_args()` (`pl_bolts.models.detection.faster_rcnn.faster_rcnn_module.FasterRCNNModule` static method), 276
`add_model_specific_args()` (`pl_bolts.models.detection.faster_rcnn.faster_rcnn.FasterRCNN` static method), 275
`add_model_specific_args()` (`pl_bolts.models.gans.basic.basic_gan_module.GAN` static method), 280
`add_model_specific_args()` (`pl_bolts.models.gans.DCGAN` static method), 279
`add_model_specific_args()` (`pl_bolts.models.gans.dcgan.dcgan_module.DCGAN` static method), 283
`add_model_specific_args()` (`pl_bolts.models.gans.dcgan.dcgan_module.DCGANModule` static method), 277
`add_model_specific_args()` (`pl_bolts.models.gans.ImageGPT` static method), 264
`add_model_specific_args()` (`pl_bolts.models.LinearRegression` static method), 260
`add_model_specific_args()` (`pl_bolts.models.LitMNIST` static method), 259
`add_model_specific_args()` (`pl_bolts.models.LogisticRegression` static method), 260
`add_model_specific_args()` (`pl_bolts.models.mnist_module.LitMNIST` static method), 382
`add_model_specific_args()` (`pl_bolts.models.regression.linear_regression.LinearRegression` static method), 285
`add_model_specific_args()` (`pl_bolts.models.regression.LinearRegression` static method), 283
`add_model_specific_args()` (`pl_bolts.models.regression.logistic_regression.LogisticRegression` static method), 286
`add_model_specific_args()` (`pl_bolts.models.regression.LogisticRegression` static method), 284
`add_model_specific_args()` (`pl_bolts.models.rl.DQN` static method), 289
`add_model_specific_args()` (`pl_bolts.models.rl.dqn_model.DQN` static method), 311
`add_model_specific_args()` (`pl_bolts.models.rl.Reinforce` static method), 296
`add_model_specific_args()` (`pl_bolts.models.rl.Reinforce` static method), 296

`(pl_bolts.models.rl.reinforce_model.Reinforce static method)`, 319
`add_model_specific_args()`
`(pl_bolts.models.rl.vanilla_policy_gradient_model.VanillaPolicyGradient static method)`, 321
`add_model_specific_args()`
`(pl_bolts.models.rl.VanillaPolicyGradient static method)`, 298
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.AMDIM static method)`, 323
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.amdim.AMDIM static method)`, 334
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.amdim.amdim_module.AMDIM static method)`, 338
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.BYOL static method)`, 325
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.byol.byol_module.BYOL static method)`, 344
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.cpc.cpc_module.CPCV2 static method)`, 349
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.cpc.CPCV2 static method)`, 345
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.CPCV2 static method)`, 326
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.moco.moco2_module.MocoV2 static method)`, 356
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.MocoV2 static method)`, 327
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.SimCLR static method)`, 328
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.simclr.simclr_module.SimCLR static method)`, 359
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.SimSiam static method)`, 330
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.simsiam.simsiam_module.SimSiam static method)`, 363
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.SwAV static method)`, 332
`add_model_specific_args()`

`(pl_bolts.models.self_supervised.swav.SwAV static method)`, 365
`add_model_specific_args()`
`(pl_bolts.models.self_supervised.swav.swav_module.SwAV static method)`, 368
`add_model_specific_args()`
`(pl_bolts.models.SemSegment static method)`, 261
`add_model_specific_args()`
`(pl_bolts.models.VAE static method)`, 259
`add_model_specific_args()`
`(pl_bolts.models.vision.image_gpt.igpt_module.ImageGPT static method)`, 379
`add_model_specific_args()`
`(pl_bolts.models.vision.segmentation.SemSegment static method)`, 381
`add_model_specific_args()`
`(pl_bolts.models.vision.SemSegment static method)`, 376
`adv_val()` `(pl_bolts.models.rl.common.networks.DuelingCNN method)`, 306
`adv_val()` `(pl_bolts.models.rl.common.networks.DuelingMLP method)`, 306
`AE` (class in `pl_bolts.models`), 257
`AE` (class in `pl_bolts.models.autoencoders`), 265
`AE` (class in `pl_bolts.models.autoencoders.basic_ae.basic_ae_module`), 267
`Agent` (class in `pl_bolts.models.rl.common.agents`), 299
`AMDIM` (class in `pl_bolts.models.self_supervised`), 322
`AMDIM` (class in `pl_bolts.models.self_supervised.amdim`), 333
`AMDIM` (class in `pl_bolts.models.self_supervised.amdim.amdim_module`), 337
`AMDIMEncoder` (class in `pl_bolts.models.self_supervised.amdim`), 334
`AMDIMEncoder` (class in `pl_bolts.models.self_supervised.amdim.networks`), 339
`AMDIMEvalTransformsCIFAR10` (class in `pl_bolts.models.self_supervised.amdim`), 334
`AMDIMEvalTransformsCIFAR10` (class in `pl_bolts.models.self_supervised.amdim.transforms`), 340
`AMDIMEvalTransformsImageNet128` (class in `pl_bolts.models.self_supervised.amdim`), 335
`AMDIMEvalTransformsImageNet128` (class in `pl_bolts.models.self_supervised.amdim.transforms`), 341
`AMDIMEvalTransformsSTL10` (class in `pl_bolts.models.self_supervised.amdim`), 335
`AMDIMEvalTransformsSTL10` (class in

`pl_bolts.models.self_supervised.amdim.transforms` `BatchGradientVerificationCallback` (class in `pl_bolts.callbacks`), 177

`AmdimNCELoss` (class in `BatchGradientVerificationCallback` (class in `pl_bolts.callbacks`), 177), 385

`AMDIMPatchesPretraining` (class in `BinaryMNIST` (class in `pl_bolts.datasets`), 248), 339

`AMDIMPretraining` (class in `BinaryMNISTDataModule` (class in `pl_bolts.datamodules`), 197), 339

`AMDIMTrainTransformsCIFAR10` (class in `pl_bolts.datamodules.binary_mnist_datamodule`), 336

`AMDIMTrainTransformsCIFAR10` (class in `Block` (class in `pl_bolts.models.vision.image_gpt.gpt2`), 377), 341

`AMDIMTrainTransformsImageNet128` (class in `Bottleneck` (class in `pl_bolts.models.self_supervised.swav.swav_resnet`), 369), 336

`AMDIMTrainTransformsImageNet128` (class in `Buffer` (class in `pl_bolts.models.rl.common.memory`), 302), 342

`AMDIMTrainTransformsSTL10` (class in `BufferWrapper` (class in `pl_bolts.models.rl.common.gym_wrappers`), 301), 336

`AMDIMTrainTransformsSTL10` (class in `build_networks()` (`pl_bolts.models.rl.DQN` method), 289), 342

`append()` (`pl_bolts.models.rl.common.memory.Buffer` method), 302

`append()` (`pl_bolts.models.rl.common.memory.MultiStepBuffer` method), 303

`append()` (`pl_bolts.models.rl.common.memory.PERBuffer` method), 304

`AsynchronousLoader` (class in `build_networks()` (`pl_bolts.models.rl.NoisyDQN` method), 293), 196

`AsynchronousLoader` (class in `BYOL` (class in `pl_bolts.models.self_supervised`), 324), 217

`BYOLMAWeightUpdate` (class in `pl_bolts.callbacks`), 178

`backward()` (`pl_bolts.models.self_supervised.simclr.simclr_module.SimCLRFunction` (class in `pl_bolts.callbacks.byol_updates`), 189), 359

`BASE_URL` (`pl_bolts.datasets.CIFAR10` attribute), 244

`BASE_URL` (`pl_bolts.datasets.cifar10_dataset.CIFAR10` attribute), 250

`BaseExperienceSource` (class in `cache_folder_name` (`pl_bolts.datasets.base_dataset.LightDataset` attribute), 249), 224

`BasicBlock` (class in `cache_folder_name` (`pl_bolts.datasets.CIFAR10` attribute), 244), 369

`BatchGradientVerification` (class in `cache_folder_name` (`pl_bolts.datasets.cifar10_dataset.CIFAR10` attribute), 250), 185

<code>cached_folder_path()</code> (<code>pl_bolts.datasets.base_dataset.LightDataset</code> property), 249	<code>pl_bolts.models.regression.linear_regression</code>), 285
<code>cached_folder_path()</code> (<code>pl_bolts.datasets.LightDataset</code> property), 242	<code>cli_main()</code> (in module <code>pl_bolts.models.regression.logistic_regression</code>), 286
<code>calc_qvals()</code> (<code>pl_bolts.models.rl.Reinforce</code> method), 296	<code>cli_main()</code> (in module <code>pl_bolts.models.rl.double_dqn_model</code>), 309
<code>calc_qvals()</code> (<code>pl_bolts.models.rl.reinforce_model.Reinforce</code> method), 319	<code>cli_main()</code> (in module <code>pl_bolts.models.rl.dqn_model</code>), 312
<code>check()</code> (<code>pl_bolts.callbacks.verifcation.base.VerificationBase</code> method), 184	<code>cli_main()</code> (in module <code>pl_bolts.models.rl.dueling_dqn_model</code>), 314
<code>check()</code> (<code>pl_bolts.callbacks.verifcation.batch_gradient.BatchGradientVerification</code> method), 185	<code>cli_main()</code> (in module <code>pl_bolts.models.rl.noisy_dqn_model</code>), 316
<code>CIFAR10</code> (class in <code>pl_bolts.datasets</code>), 243	<code>cli_main()</code> (in module <code>pl_bolts.models.rl.per_dqn_model</code>), 317
<code>CIFAR10</code> (class in <code>pl_bolts.datasets.cifar10_dataset</code>), 249	<code>cli_main()</code> (in module <code>pl_bolts.models.rl.reinforce_model</code>), 320
<code>cifar10()</code> (<code>pl_bolts.models.self_supervised.amdim.datasets.AMDIMPretraining</code> static method), 339	<code>cli_main()</code> (in module <code>pl_bolts.models.rl.vanilla_policy_gradient_model</code>), 321
<code>cifar10()</code> (<code>pl_bolts.models.self_supervised.amdim.datasets.AMDIMPretraining</code> static method), 339	<code>cli_main()</code> (in module <code>pl_bolts.models.self_supervised.amdim.amdim_module</code>), 338
<code>cifar10_normalization()</code> (in module <code>pl_bolts.transforms.dataset_normalizations</code>), 391	<code>cli_main()</code> (in module <code>pl_bolts.models.self_supervised.amdim.amdim_module</code>), 344
<code>cifar10_tiny()</code> (<code>pl_bolts.models.self_supervised.amdim.datasets.AMDIMPretraining</code> static method), 339	<code>cli_main()</code> (in module <code>pl_bolts.models.self_supervised.byol.byol_module</code>), 344
<code>CIFAR10DataModule</code> (class in <code>pl_bolts.datamodules</code>), 198	<code>cli_main()</code> (in module <code>pl_bolts.models.self_supervised.cpc.cpc_finetuner</code>), 349
<code>CIFAR10DataModule</code> (class in <code>pl_bolts.datamodules.cifar10_datamodule</code>), 219	<code>cli_main()</code> (in module <code>pl_bolts.models.self_supervised.moco.moco2_module</code>), 356
<code>CIFAR10Mixed</code> (class in <code>pl_bolts.datasets</code>), 248	<code>cli_main()</code> (in module <code>pl_bolts.models.self_supervised.simclr.simclr_finetuner</code>), 358
<code>CIFAR10Mixed</code> (class in <code>pl_bolts.datasets.ssl_amdim_datasets</code>), 256	<code>cli_main()</code> (in module <code>pl_bolts.models.self_supervised.simclr.simclr_module</code>), 359
<code>CityscapesDataModule</code> (class in <code>pl_bolts.datamodules</code>), 201	<code>cli_main()</code> (in module <code>pl_bolts.models.self_supervised.simsiam.simsiam_module</code>), 364
<code>CityscapesDataModule</code> (class in <code>pl_bolts.datamodules.cityscapes_datamodule</code>), 222	<code>cli_main()</code> (in module <code>pl_bolts.models.self_supervised.swav.swav_finetuner</code>), 367
<code>cli_main()</code> (in module <code>pl_bolts.models.autoencoders.basic_ae.basic_ae_module</code>), 268	<code>cli_main()</code> (in module <code>pl_bolts.models.self_supervised.swav.swav_module</code>), 369
<code>cli_main()</code> (in module <code>pl_bolts.models.autoencoders.basic_vae.basic_vae_module</code>), 270	<code>cli_main()</code> (in module <code>pl_bolts.models.gans.basic.basic_gan_module</code>), 280
<code>cli_main()</code> (in module <code>pl_bolts.models.gans.basic.basic_gan_module</code>), 280	<code>cli_main()</code> (in module <code>pl_bolts.models.vision.image_gpt.igpt_module</code>), 380
<code>cli_main()</code> (in module <code>pl_bolts.models.gans.dcgan.dcgan_module</code>), 283	<code>cli_main()</code> (in module <code>pl_bolts.models.vision.segmentation</code>), 381
<code>cli_main()</code> (in module <code>pl_bolts.models.mnist_module</code>), 382	<code>CNN</code> (class in <code>pl_bolts.models.rl.common.networks</code>), 305
<code>cli_main()</code> (in module <code>pl_bolts.models.mnist_module</code>), 382	<code>collect_and_name_tensors()</code> (in module <code>pl_bolts.models.vision.segmentation</code>), 381

`pl_bolts.callbacks.data_monitor`), 192
`collect_tensors()` (in module `pl_bolts.callbacks.data_monitor`), 192
`pl_bolts.callbacks.data_monitor`), 192
`pl_bolts.callbacks.data_monitor`), 192
`Compose` (class in `pl_bolts.datamodules.vocdetetection_datamodule`), 260
`compute_loss_h()` (`pl_bolts.losses.self_supervised_learning.CPC70k` method), 385
`compute_returns()` (`pl_bolts.models.rl.vanilla_policy_gradient_model.VanillaPolicyGradient` method), 321
`compute_returns()` (`pl_bolts.models.rl.VanillaPolicyGradient` method), 298
`concat_all_gather()` (in module `pl_bolts.models.self_supervised.moco.moco2_module`), 356
`ConcatDataset` (class in `pl_bolts.datasets`), 245
`ConcatDataset` (class in `pl_bolts.datasets.concat_dataset`), 251
`configure_optimizers()` (`pl_bolts.models.AE` method), 258
`configure_optimizers()` (`pl_bolts.models.autoencoders.AE` method), 265
`configure_optimizers()` (`pl_bolts.models.autoencoders.basic_ae.basic_ae_module.AE` method), 268
`configure_optimizers()` (`pl_bolts.models.autoencoders.basic_vae.basic_vae_module.VAE` method), 269
`configure_optimizers()` (`pl_bolts.models.autoencoders.VAE` method), 266
`configure_optimizers()` (`pl_bolts.models.detection.faster_rcnn.faster_rcnn_module.FasterRCNN` method), 276
`configure_optimizers()` (`pl_bolts.models.detection.faster_rcnn.FasterRCNN` method), 275
`configure_optimizers()` (`pl_bolts.models.detection.FasterRCNN` method), 272
`configure_optimizers()` (`pl_bolts.models.gans.basic.basic_gan_module.GAN` method), 280
`configure_optimizers()` (`pl_bolts.models.gans.DCGAN` method), 279
`configure_optimizers()` (`pl_bolts.models.gans.dcgan.dcgan_module.DCGAN` method), 283
`configure_optimizers()` (`pl_bolts.models.gans.GAN` method), 277
`configure_optimizers()`
`configure_optimizers()` (`pl_bolts.models.ImageGPT` method), 264
`configure_optimizers()` (`pl_bolts.models.LinearRegression` method),
`configure_optimizers()` (`pl_bolts.models.LitMNIST` method), 259
`configure_optimizers()` (`pl_bolts.models.LogisticRegression` method),
`configure_optimizers()` (`pl_bolts.models.mnist_module.LitMNIST` method), 382
`configure_optimizers()` (`pl_bolts.models.regression.linear_regression.LinearRegression` method), 285
`configure_optimizers()` (`pl_bolts.models.regression.LinearRegression` method), 283
`configure_optimizers()` (`pl_bolts.models.regression.logistic_regression.LogisticRegression` method), 286
`configure_optimizers()` (`pl_bolts.models.regression.LogisticRegression` method), 284
`configure_optimizers()` (`pl_bolts.models.rl.DQN` method), 289
`configure_optimizers()` (`pl_bolts.models.rl.dqn_model.DQN` method),
`configure_optimizers()` (`pl_bolts.models.rl.Reinforce` method), 296
`configure_optimizers()` (`pl_bolts.models.rl.reinforce_model.Reinforce` method), 319
`configure_optimizers()` (`pl_bolts.models.rl.vanilla_policy_gradient_model.VanillaPolicy` method), 321
`configure_optimizers()` (`pl_bolts.models.rl.VanillaPolicyGradient` method), 298
`configure_optimizers()` (`pl_bolts.models.self_supervised.AMDIM` method), 324
`configure_optimizers()` (`pl_bolts.models.self_supervised.amdim.AMDIM` method), 334
`configure_optimizers()` (`pl_bolts.models.self_supervised.amdim.amdim_module.AMDIM` method), 338
`configure_optimizers()` (`pl_bolts.models.self_supervised.BYOL` method), 325
`configure_optimizers()`

```

(pl_bolts.models.self_supervised.byol.byol_module.Byol module), 344
configure_optimizers() (pl_bolts.models.self_supervised.cpc.cpc_module.CPC module), 350
configure_optimizers() (pl_bolts.models.self_supervised.cpc.CPCV2 module), 345
configure_optimizers() (pl_bolts.models.self_supervised.CPCV2 module), 326
configure_optimizers() (pl_bolts.models.self_supervised.moco.moco2_module.MocoV2 module), 356
configure_optimizers() (pl_bolts.models.self_supervised.MocoV2 module), 327
configure_optimizers() (pl_bolts.models.self_supervised.SimCLR module), 328
configure_optimizers() (pl_bolts.models.self_supervised.simclr.simclr_module.SimCLR module), 359
configure_optimizers() (pl_bolts.models.self_supervised.SimSiam module), 330
configure_optimizers() (pl_bolts.models.self_supervised.simsiam.simsiam_module.SimSiam module), 363
configure_optimizers() (pl_bolts.models.self_supervised.ssl_finetuner.SSLFineTuner module), 375
configure_optimizers() (pl_bolts.models.self_supervised.SSLFineTuner module), 331
configure_optimizers() (pl_bolts.models.self_supervised.SwAV module), 332
configure_optimizers() (pl_bolts.models.self_supervised.swav.SwAV module), 365
configure_optimizers() (pl_bolts.models.self_supervised.swav.swav_module.SwAV module), 368
configure_optimizers() (pl_bolts.models.SemSegment method), 261
configure_optimizers() (pl_bolts.models.VAE module), 259
configure_optimizers() (pl_bolts.models.vision.image_gpt.igpt_module.ImageGPT module), 379
configure_optimizers() (pl_bolts.models.vision.segmentation.SemSegment method), 381

configure_optimizers() (pl_bolts.models.vision.SemSegment module), 376
CPCV2UsedLogitCallback (class in pl_bolts.callbacks), 181
ConfusedLogitCallback (class in pl_bolts.callbacks.vision.confused_logit), 187
conv1x1() (in module pl_bolts.models.autoencoders.components), 271
conv1x1() (in module pl_bolts.models.self_supervised.cpc.networks), 350
conv1x1() (in module pl_bolts.models.self_supervised.swav.swav_resnet), 370
Conv3x3 (class in pl_bolts.models.self_supervised.amdim.networks), 339
conv3x3() (in module pl_bolts.models.autoencoders.components), 271
conv3x3() (in module pl_bolts.models.self_supervised.cpc.networks), 350
conv3x3() (in module pl_bolts.models.self_supervised.swav.swav_resnet), 370
conv_block() (pl_bolts.models.PixelCNN module), 261
conv_block() (pl_bolts.models.vision.pixel_cnn.PixelCNN module), 380
conv_block() (pl_bolts.models.vision.PixelCNN module), 375
ConvResBlock (class in pl_bolts.models.self_supervised.amdim.networks), 339
ConvResNxN (class in pl_bolts.models.self_supervised.amdim.networks), 340
cosine_similarity() (pl_bolts.models.self_supervised.SimSiam module), 330
cosine_similarity() (pl_bolts.models.self_supervised.simsiam.simsiam_module.SimSiam module), 363
cpc_resnet101() (in module pl_bolts.models.self_supervised.cpc), 349
cpc_resnet101() (in module pl_bolts.models.self_supervised.cpc.networks), 350
cpc_resnet50() (in module pl_bolts.models.self_supervised.cpc), 349
cpc_resnet50() (in module

```

`pl_bolts.models.self_supervised.cpc.networks)`,
 350
`CPCEvalTransformsCIFAR10` (class in
`pl_bolts.models.self_supervised.cpc`), 345
`CPCEvalTransformsCIFAR10` (class in
`pl_bolts.models.self_supervised.cpc.transforms)`,
 351
`CPCEvalTransformsImageNet128` (class in
`pl_bolts.models.self_supervised.cpc`), 346
`CPCEvalTransformsImageNet128` (class in
`pl_bolts.models.self_supervised.cpc.transforms)`,
 351
`CPCEvalTransformsSTL10` (class in
`pl_bolts.models.self_supervised.cpc`), 346
`CPCEvalTransformsSTL10` (class in
`pl_bolts.models.self_supervised.cpc.transforms)`,
 352
`CPCResNet` (class in
`pl_bolts.models.self_supervised.cpc.networks)`,
 350
`CPCTask` (class in `pl_bolts.losses.self_supervised_learning`),
 385
`CPCTrainTransformsCIFAR10` (class in
`pl_bolts.models.self_supervised.cpc`), 347
`CPCTrainTransformsCIFAR10` (class in
`pl_bolts.models.self_supervised.cpc.transforms)`,
 352
`CPCTrainTransformsImageNet128` (class in
`pl_bolts.models.self_supervised.cpc`), 347
`CPCTrainTransformsImageNet128` (class in
`pl_bolts.models.self_supervised.cpc.transforms)`,
 353
`CPCTrainTransformsSTL10` (class in
`pl_bolts.models.self_supervised.cpc`), 348
`CPCTrainTransformsSTL10` (class in
`pl_bolts.models.self_supervised.cpc.transforms)`,
 353
`CPCV2` (class in `pl_bolts.models.self_supervised`), 325
`CPCV2` (class in `pl_bolts.models.self_supervised.cpc`),
 345
`CPCV2` (class in `pl_bolts.models.self_supervised.cpc.cpc_module`),
 349
`create_fasterrcnn_backbone()` (in module
`pl_bolts.models.detection.faster_rcnn`), 275
`create_fasterrcnn_backbone()` (in module
`pl_bolts.models.detection.faster_rcnn.backbones`),
 275
`create_torchvision_backbone()` (in module
`pl_bolts.models.detection.components`), 273
`create_torchvision_backbone()` (in module
`pl_bolts.models.detection.components.torchvision_backbones`),
 273
D
`data` (`pl_bolts.datasets.base_dataset.LightDataset` at-
 tribute), 249
`data` (`pl_bolts.datasets.CIFAR10` attribute), 244
`data` (`pl_bolts.datasets.cifar10_dataset.CIFAR10`
 attribute), 250
`data` (`pl_bolts.datasets.cifar10_dataset.TrialCIFAR10`
 attribute), 251
`data` (`pl_bolts.datasets.LightDataset` attribute), 243
`data` (`pl_bolts.datasets.TrialCIFAR10` attribute), 245
`DataAugmentation` (class in
`pl_bolts.models.rl.common.gym_wrappers`),
 301
`DataMonitorBase` (class in
`pl_bolts.callbacks.data_monitor`), 190
`dataset_cls` (`pl_bolts.datamodules.binary_mnist_datamodule.BinaryM`
 attribute), 219
`dataset_cls` (`pl_bolts.datamodules.BinaryMNISTDataModule`
 attribute), 198
`dataset_cls` (`pl_bolts.datamodules.cifar10_datamodule.CIFAR10Data`
 attribute), 221
`dataset_cls` (`pl_bolts.datamodules.cifar10_datamodule.TinyCIFAR10D`
 attribute), 222
`dataset_cls` (`pl_bolts.datamodules.CIFAR10DataModule`
 attribute), 200
`dataset_cls` (`pl_bolts.datamodules.fashion_mnist_datamodule.Fashion`
 attribute), 228
`dataset_cls` (`pl_bolts.datamodules.FashionMNISTDataModule`
 attribute), 206
`dataset_cls` (`pl_bolts.datamodules.mnist_datamodule.MNISTDataMod`
 attribute), 234
`dataset_cls` (`pl_bolts.datamodules.MNISTDataModule`
 attribute), 211
`dataset_cls` (`pl_bolts.datamodules.TinyCIFAR10DataModule`
 attribute), 200
`dataset_cls` (`pl_bolts.datamodules.vision_datamodule.VisionDataMod`
 attribute), 241
`DATASET_NAME` (`pl_bolts.datasets.base_dataset.LightDataset`
 attribute), 249
`DATASET_NAME` (`pl_bolts.datasets.CIFAR10` attribute),
 244
`DATASET_NAME` (`pl_bolts.datasets.cifar10_dataset.CIFAR10`
 attribute), 250
`DATASET_NAME` (`pl_bolts.datasets.LightDataset` at-
 tribute), 242
`DCGAN` (class in `pl_bolts.models.gans`), 278
`DCGAN` (class in `pl_bolts.models.gans.dcgan.dcgan_module`),
 282
`DCGANDiscriminator` (class in
`pl_bolts.models.gans.dcgan.components`),
 281
`DCGANGenerator` (class in
`pl_bolts.models.gans.dcgan.components`),
 281

DecoderBlock (class in attribute), 198
 (pl_bolts.models.autoencoders.components), 270

DecoderBottleneck (class in dims (pl_bolts.datamodules.cifar10_datamodule.CIFAR10DataModule attribute), 221
 (pl_bolts.models.autoencoders.components), 270
 dims (pl_bolts.datamodules.cifar10_datamodule.TinyCIFAR10DataModule attribute), 222

default_input_mapping() (in module attribute), 200
 (pl_bolts.callbacks.verificaton.batch_gradient), 186
 dims (pl_bolts.datamodules.fashion_mnist_datamodule.FashionMNISTDataModule attribute), 228

default_output_mapping() (in module dims (pl_bolts.datamodules.FashionMNISTDataModule attribute), 206
 (pl_bolts.callbacks.verificaton.batch_gradient), 186
 dims (pl_bolts.datamodules.mnist_datamodule.MNISTDataModule attribute), 234

default_transforms() (pl_bolts.datamodules.binary_mnist_datamodule.BinaryMNISTDataModule attribute), 211
 (method), 219
 dims (pl_bolts.datamodules.TinyCIFAR10DataModule attribute), 201

default_transforms() (pl_bolts.datamodules.BinaryMNISTDataModule attribute), 198
 dims (pl_bolts.datamodules.vision_datamodule.VisionDataModule attribute), 241

default_transforms() (pl_bolts.datamodules.cifar10_datamodule.CIFAR10DataModule attribute), 221
 (method), 221
 (pl_bolts.datasets.base_dataset.LightDataset attribute), 249

default_transforms() (pl_bolts.datamodules.CIFAR10DataModule attribute), 200
 dir_path (pl_bolts.datasets.CIFAR10 attribute), 244
 dir_path (pl_bolts.datasets.cifar10_dataset.CIFAR10 attribute), 250

default_transforms() (pl_bolts.datamodules.fashion_mnist_datamodule.FashionMNISTDataModule attribute), 228
 dir_path (pl_bolts.datasets.cifar10_dataset.TinyCIFAR10 attribute), 243
 dir_path (pl_bolts.datasets.LightDataset attribute), 243

default_transforms() (pl_bolts.datamodules.FashionMNISTDataModule attribute), 205
 dir_path (pl_bolts.datasets.TinyCIFAR10 attribute), 245

default_transforms() (pl_bolts.datamodules.mnist_datamodule.MNISTDataModule attribute), 234
 discount_rewards() (pl_bolts.datamodules.DiscountedExperienceSource method), 203

default_transforms() (pl_bolts.datamodules.MNISTDataModule attribute), 210
 discount_rewards() (pl_bolts.datamodules.experience_source.DiscountedExperienceSource method), 224

default_transforms() (pl_bolts.datamodules.vision_datamodule.VisionDataModule attribute), 240
 discount_rewards() (pl_bolts.models.rl.common.memory.MultiStepBuffer method), 304

defaults() (pl_bolts.optimizers.lars_scheduling.LARSWrapper attribute), 389
 discount_rewards() (pl_bolts.models.rl.Reinforce method), 296

defaults() (pl_bolts.optimizers.LARSWrapper property), 387
 discount_rewards() (pl_bolts.models.rl.reinforce_model.Reinforce method), 319

deterministic_shuffle() (pl_bolts.datasets.ssl_anddim_datasets.SSLDatasetMixin attribute), 256
 DiscountedExperienceSource (class in pl_bolts.datamodules), 202

deterministic_shuffle() (pl_bolts.datasets.SSLDatasetMixin class method), 248
 DiscountedExperienceSource (class in pl_bolts.datamodules.experience_source), 224

dicts_to_table() (in module Discriminator (class in pl_bolts.models.gans.basic.components), 194

dims (pl_bolts.datamodules.binary_mnist_datamodule.BinaryMNISTDataModule attribute), 219
 discriminator_loss() (pl_bolts.models.gans.basic.basic_gan_module.GAN

`method`), 280
`discriminator_loss()` (*pl_bolts.models.gans.GAN method*), 277
`discriminator_step()` (*pl_bolts.models.gans.basic.basic_gan_module.GAN method*), 280
`discriminator_step()` (*pl_bolts.models.gans.GAN method*), 277
`distributed_sinkhorn()` (*pl_bolts.models.self_supervised.SwAV method*), 332
`distributed_sinkhorn()` (*pl_bolts.models.self_supervised.swav.SwAV method*), 365
`distributed_sinkhorn()` (*pl_bolts.models.self_supervised.swav.swav_module.SwAV method*), 368
`done()` (*pl_bolts.datamodules.experience_source.ExperienceSource property*), 225
`done()` (*pl_bolts.models.rl.common.memory.ExperienceMemory property*), 303
`double_dqn_loss()` (in module *pl_bolts.losses.rl*), 384
`DoubleConv` (class in *pl_bolts.models.vision.unet*), 381
`DoubleDQN` (class in *pl_bolts.models.rl*), 286
`DoubleDQN` (class in *pl_bolts.models.rl.double_dqn_model*), 308
`Down` (class in *pl_bolts.models.vision.unet*), 381
`download()` (*pl_bolts.datasets.CIFAR10 method*), 244
`download()` (*pl_bolts.datasets.cifar10_dataset.CIFAR10 method*), 250
`DQN` (class in *pl_bolts.models.rl*), 288
`DQN` (class in *pl_bolts.models.rl.dqn_model*), 310
`dqn_loss()` (in module *pl_bolts.losses.rl*), 384
`DuelingCNN` (class in *pl_bolts.models.rl.common.networks*), 306
`DuelingDQN` (class in *pl_bolts.models.rl*), 290
`DuelingDQN` (class in *pl_bolts.models.rl.dueling_dqn_model*), 313
`DuelingMLP` (class in *pl_bolts.models.rl.common.networks*), 306
`DummyDataset` (class in *pl_bolts.datasets*), 245
`DummyDataset` (class in *pl_bolts.datasets.dummy_dataset*), 252
`DummyDetectionDataset` (class in *pl_bolts.datasets*), 245
`DummyDetectionDataset` (class in *pl_bolts.datasets.dummy_dataset*), 252

E

`encode_segmap()` (*pl_bolts.datamodules.KittiDataset method*), 217
`encode_segmap()` (*pl_bolts.datasets.kitti_dataset.KittiDataset method*), 255
`encode_segmap()` (*pl_bolts.datasets.KittiDataset method*), 248
`EncoderBlock` (class in *pl_bolts.models.autoencoders.components*), 270
`EncoderBottleneck` (class in *pl_bolts.models.autoencoders.components*), 270
`env_actions()` (*pl_bolts.datamodules.experience_source.ExperienceSource method*), 226
`env_actions()` (*pl_bolts.datamodules.ExperienceSource method*), 203
`env_step()` (*pl_bolts.datamodules.experience_source.ExperienceSource method*), 226
`env_step()` (*pl_bolts.datamodules.ExperienceSource method*), 203
`error_message()` (*pl_bolts.callbacks.verfication.base.VerificationCallback method*), 184
`exclude_from_wt_decay()` (*pl_bolts.models.self_supervised.SimCLR method*), 328
`exclude_from_wt_decay()` (*pl_bolts.models.self_supervised.simclr.simclr_module.SimCLR method*), 359
`exclude_from_wt_decay()` (*pl_bolts.models.self_supervised.SimSiam method*), 330
`exclude_from_wt_decay()` (*pl_bolts.models.self_supervised.simsiam.simsiam_module.SimSiam method*), 363
`exclude_from_wt_decay()` (*pl_bolts.models.self_supervised.SwAV method*), 332
`exclude_from_wt_decay()` (*pl_bolts.models.self_supervised.swav.SwAV method*), 365
`exclude_from_wt_decay()` (*pl_bolts.models.self_supervised.swav.swav_module.SwAV method*), 368
`expansion` (*pl_bolts.models.autoencoders.components.DecoderBlock attribute*), 270
`expansion` (*pl_bolts.models.autoencoders.components.DecoderBottleneck attribute*), 270
`expansion` (*pl_bolts.models.autoencoders.components.EncoderBlock attribute*), 270
`expansion` (*pl_bolts.models.autoencoders.components.EncoderBottleneck attribute*), 270
`expansion` (*pl_bolts.models.self_supervised.swav.swav_resnet.BasicBlock attribute*), 369
`expansion` (*pl_bolts.models.self_supervised.swav.swav_resnet.Bottleneck attribute*), 369
`Experience` (class in *pl_bolts.datamodules.experience_source*), 225

Experience (class in `pl_bolts.models.rl.common.memory`), 302

ExperienceSource (class in `pl_bolts.datamodules`), 203

ExperienceSource (class in `pl_bolts.datamodules.experience_source`), 225

ExperienceSourceDataset (class in `pl_bolts.datamodules`), 204

ExperienceSourceDataset (class in `pl_bolts.datamodules.experience_source`), 227

extra_args (`pl_bolts.datamodules.cityscapes_datamodule.CityscapesDataModule` attribute), 224

extra_args (`pl_bolts.datamodules.CityscapesDataModule` attribute), 202

EXTRA_ARGS (`pl_bolts.datamodules.vision_datamodule.VisionDataModule` attribute), 241

extract_archive() (in module `pl_bolts.datasets`), 248

extract_archive() (in module `pl_bolts.datasets.imagenet_dataset`), 254

F

FakeRKHSConvNet (class in `pl_bolts.models.self_supervised.amdim.networks`), 340

FashionMNISTDataModule (class in `pl_bolts.datamodules`), 204

FashionMNISTDataModule (class in `pl_bolts.datamodules.fashion_mnist_datamodule`), 227

FasterRCNN (class in `pl_bolts.models.detection`), 272

FasterRCNN (class in `pl_bolts.models.detection.faster_rcnn`), 274

FasterRCNN (class in `pl_bolts.models.detection.faster_rcnn.faster_rcnn_module`), 276

feat_size_w_mask() (`pl_bolts.losses.self_supervised_learning.FeatureMapContrastiveTask` method), 386

FeatureMapContrastiveTask (class in `pl_bolts.losses.self_supervised_learning`), 385

FILE_NAME (`pl_bolts.datasets.CIFAR10` attribute), 244

FILE_NAME (`pl_bolts.datasets.cifar10_dataset.CIFAR10` attribute), 250

FireResetEnv (class in `pl_bolts.models.rl.common.gym_wrappers`), 301

Flatten (class in `pl_bolts.models.self_supervised.evaluators`), 371

flatten() (`pl_bolts.models.self_supervised.cpc.networks.CPCResNet` method), 350

forward() (`pl_bolts.losses.self_supervised_learning.AmdimNCELoss` method), 385

forward() (`pl_bolts.losses.self_supervised_learning.CPCTask` method), 385

forward() (`pl_bolts.losses.self_supervised_learning.FeatureMapContrastiveTask` method), 386

forward() (`pl_bolts.models.AE` method), 258

forward() (`pl_bolts.models.autoencoders.AE` method), 265

forward() (`pl_bolts.models.autoencoders.basic_ae.basic_ae_module.AE` method), 268

forward() (`pl_bolts.models.autoencoders.basic_vae.basic_vae_module.AE` method), 269

forward() (`pl_bolts.models.autoencoders.components.DecoderBlock` method), 270

forward() (`pl_bolts.models.autoencoders.components.DecoderBottleneck` method), 270

forward() (`pl_bolts.models.autoencoders.components.EncoderBlock` method), 270

forward() (`pl_bolts.models.autoencoders.components.EncoderBottleneck` method), 270

forward() (`pl_bolts.models.autoencoders.components.Interpolate` method), 271

forward() (`pl_bolts.models.autoencoders.components.ResNetDecoder` method), 271

forward() (`pl_bolts.models.autoencoders.components.ResNetEncoder` method), 271

forward() (`pl_bolts.models.autoencoders.VAE` method), 266

forward() (`pl_bolts.models.detection.faster_rcnn.faster_rcnn_module.FasterRCNN` method), 277

forward() (`pl_bolts.models.detection.faster_rcnn.FasterRCNN` method), 275

forward() (`pl_bolts.models.detection.FasterRCNN` method), 272

forward() (`pl_bolts.models.gans.basic.basic_gan_module.GAN` method), 280

forward() (`pl_bolts.models.gans.basic.components.Discriminator` method), 281

forward() (`pl_bolts.models.gans.basic.components.Generator` method), 281

forward() (`pl_bolts.models.gans.DCGAN` method), 279

forward() (`pl_bolts.models.gans.dcgan.components.DCGANDiscriminator` method), 281

forward() (`pl_bolts.models.gans.dcgan.components.DCGANGenerator` method), 281

forward() (`pl_bolts.models.gans.dcgan.dcgan_module.DCGAN` method), 283

forward() (`pl_bolts.models.gans.GAN` method), 277

forward() (`pl_bolts.models.GPT2` method), 262

forward() (`pl_bolts.models.ImageGPT` method), 264

forward() (`pl_bolts.models.LinearRegression` method), 260

forward() (pl_bolts.models.self_supervised.swav.swav_module.SwAV method), 368	from_pretrained() (pl_bolts.models.VAE method), 258
forward() (pl_bolts.models.self_supervised.swav.swav_resnet.BasicBlock method), 369	
forward() (pl_bolts.models.self_supervised.swav.swav_resnet.Bottleneck method), 369	
forward() (pl_bolts.models.self_supervised.swav.swav_resnet.MultiPrototypesGAN (class in pl_bolts.models.gans), 277	
forward() (pl_bolts.models.self_supervised.swav.swav_resnet.MultiPrototypesGAN (class in pl_bolts.models.gans.basic.basic_gan_module), 279	
forward() (pl_bolts.models.self_supervised.swav.swav_resnet.ResNet (class in pl_bolts.models.gans), 277	
forward() (pl_bolts.models.self_supervised.swav.swav_resnet.ResNet (class in pl_bolts.models.self_supervised.moco.transforms), 357	
forward() (pl_bolts.models.SemSegment method), 261	
forward() (pl_bolts.models.UNet method), 262	
forward() (pl_bolts.models.VAE method), 259	
forward() (pl_bolts.models.vision.image_gpt.gpt2.Block method), 377	
forward() (pl_bolts.models.vision.image_gpt.gpt2.GPT2 method), 377	
forward() (pl_bolts.models.vision.image_gpt.igpt_module.ImageGPT method), 379	
forward() (pl_bolts.models.vision.pixel_cnn.PixelCNN method), 380	
forward() (pl_bolts.models.vision.PixelCNN method), 375	
forward() (pl_bolts.models.vision.segmentation.SemSegment method), 381	
forward() (pl_bolts.models.vision.SemSegment method), 376	
forward() (pl_bolts.models.vision.UNet method), 376	
forward() (pl_bolts.models.vision.unet.DoubleConv method), 381	
forward() (pl_bolts.models.vision.unet.Down method), 381	
forward() (pl_bolts.models.vision.unet.UNet method), 382	
forward() (pl_bolts.models.vision.unet.Up method), 382	
forward_backbone() (pl_bolts.models.self_supervised.swav.swav_resnet.ResNet method), 370	
forward_head() (pl_bolts.models.self_supervised.swav.swav_resnet.ResNet method), 370	
from_pretrained() (pl_bolts.models.AE method), 258	
from_pretrained() (pl_bolts.models.autoencoders.AE method), 265	
from_pretrained() (pl_bolts.models.autoencoders.basic_ae.basic_ae_module.AE method), 268	
from_pretrained() (pl_bolts.models.autoencoders.basic_vae.basic_vae_module.VAE method), 269	
from_pretrained() (pl_bolts.models.autoencoders.VAE method), 268	
	generate_meta_bins() (pl_bolts.datasets.imagenet_dataset.UnlabeledImagenet class method), 254
	generate_meta_bins() (pl_bolts.datasets.UnlabeledImagenet class method), 247
	generate_train_val_split() (pl_bolts.datasets.ssl_amdim_datasets.SSLDatasetMixin class method), 256
	generate_train_val_split() (pl_bolts.datasets.SSLDatasetMixin class method), 248
	Generator (class in pl_bolts.models.gans.basic.components), 281
	generator_loss() (pl_bolts.models.gans.basic.basic_gan_module.GAN method), 280
	generator_loss() (pl_bolts.models.gans.GAN method), 278
	generator_step() (pl_bolts.models.gans.basic.basic_gan_module.GAN method), 280
	generator_step() (pl_bolts.models.gans.GAN method), 278
	get_action() (pl_bolts.models.rl.common.agents.ValueAgent method), 300
	get_all_representations() (pl_bolts.callbacks.knn_online.KNNOnlineEvaluator method), 193
	get_dataset() (pl_bolts.models.self_supervised.amdim.datasets.AMDI static method), 339
	get_device() (pl_bolts.models.rl.Reinforce method), 296
	get_device() (pl_bolts.models.rl.reinforce_model.Reinforce method), 319
	get_device() (pl_bolts.models.rl.vanilla_policy_gradient_model.Vanilla method), 321

`get_device()` (*pl_bolts.models.rl.VanillaPolicyGradientImageGPT* (class in *pl_bolts.models.vision.image_gpt.igpt_module*), method), 298
`get_filenames()` (*pl_bolts.datamodules.KittiDataset* (class in *pl_bolts.datamodules.kitti_dataset.kitti_dataset_module*), method), 217
`get_filenames()` (*pl_bolts.datasets.kitti_dataset.KittiDataset* (class in *pl_bolts.datasets.kitti_dataset.kitti_dataset_module*), method), 255
`get_filenames()` (*pl_bolts.datasets.KittiDataset* (class in *pl_bolts.datasets.kitti_dataset.kitti_dataset_module*), method), 248
`get_lr()` (*pl_bolts.optimizers.LinearWarmupCosineAnnealingLR* (class in *pl_bolts.optimizers.lr_scheduler.linear_warmup_cosine_annealing_lr_module*), method), 388
`get_lr()` (*pl_bolts.optimizers.lr_scheduler.LinearWarmupCosineAnnealingLR* (class in *pl_bolts.optimizers.lr_scheduler.linear_warmup_cosine_annealing_lr_module*), method), 390
`get_random_action()` (*pl_bolts.models.rl.common.agents.ValueAgent* (class in *pl_bolts.models.rl.common.agents.value_agent_module*), method), 300
`get_representations()` (*pl_bolts.callbacks.knn_online.KNNOnlineEvaluator* (class in *pl_bolts.callbacks.knn_online.knn_online_evaluator_module*), method), 193
`get_representations()` (*pl_bolts.callbacks.ssl_online.SSLOnlineEvaluator* (class in *pl_bolts.callbacks.ssl_online.ssl_online_evaluator_module*), method), 195
`get_representations()` (*pl_bolts.callbacks.SSLOnlineEvaluator* (class in *pl_bolts.callbacks.ssl_online.ssl_online_evaluator_module*), method), 180
`giou()` (in module *pl_bolts.metrics.object_detection*), 256
`giou_loss()` (in module *pl_bolts.losses.object_detection*), 382
GPT2 (class in *pl_bolts.models*), 262
GPT2 (class in *pl_bolts.models.vision.image_gpt.gpt2*), 377
GROUP_NAME (*pl_bolts.callbacks.data_monitor.TrainingDataMonitor* (class in *pl_bolts.callbacks.data_monitor.training_data_monitor_module*), attribute), 192
GROUP_NAME (*pl_bolts.callbacks.TrainingDataMonitor* (class in *pl_bolts.callbacks.data_monitor.training_data_monitor_module*), attribute), 179
GROUP_NAME_INPUT (*pl_bolts.callbacks.data_monitor.ModuleDataMonitor* (class in *pl_bolts.callbacks.data_monitor.module_data_monitor_module*), attribute), 191
GROUP_NAME_INPUT (*pl_bolts.callbacks.ModuleDataMonitor* (class in *pl_bolts.callbacks.data_monitor.module_data_monitor_module*), attribute), 179
GROUP_NAME_OUTPUT (*pl_bolts.callbacks.data_monitor.ModuleDataMonitor* (class in *pl_bolts.callbacks.data_monitor.module_data_monitor_module*), attribute), 191
GROUP_NAME_OUTPUT (*pl_bolts.callbacks.ModuleDataMonitor* (class in *pl_bolts.callbacks.data_monitor.module_data_monitor_module*), attribute), 179
I
IMAGE_PATH (*pl_bolts.datamodules.KittiDataset* (class in *pl_bolts.datamodules.kitti_dataset.kitti_dataset_module*), attribute), 217
IMAGE_PATH (*pl_bolts.datasets.kitti_dataset.KittiDataset* (class in *pl_bolts.datasets.kitti_dataset.kitti_dataset_module*), attribute), 255
IMAGE_PATH (*pl_bolts.datasets.KittiDataset* (class in *pl_bolts.datasets.kitti_dataset.kitti_dataset_module*), attribute), 248
ImageGPT (class in *pl_bolts.models*), 262

`init_model()` (`pl_bolts.models.self_supervised.simsiam.module.SimSiam` (class in `pl_bolts.callbacks.variational`), 196
 method), 363
`init_model()` (`pl_bolts.models.self_supervised.SwAV` (class in `pl_bolts.datasets`), 242
 method), 332
`init_model()` (`pl_bolts.models.self_supervised.swav.SwAV` (class in `pl_bolts.datasets.base_dataset`), 249
 method), 365
`init_model()` (`pl_bolts.models.self_supervised.swav.swav_module.SwAV` (class in `pl_bolts.models.regression`), 283
 method), 368
`init_weights()` (`pl_bolts.models.self_supervised.amdim.AMDIMEncoder` (class in `pl_bolts.models.regression.linear_regression`),
 method), 334
`init_weights()` (`pl_bolts.models.self_supervised.amdim.networks.AMDIMEncoder` (class in `pl_bolts.models.regression.linear_regression`),
 method), 339
`init_weights()` (`pl_bolts.models.self_supervised.amdim.networks.CNNResBlock` (class in `pl_bolts.callbacks`), 387
 method), 340
`init_weights()` (`pl_bolts.models.self_supervised.amdim.networks.CNNResNet` (class in `pl_bolts.callbacks`), 389
 method), 340
`init_weights()` (`pl_bolts.models.self_supervised.amdim.networks.FakeRNNScheduler` (class in `pl_bolts.models.mnist_module`), 382
 method), 340
`Interpolate` (class in `pl_bolts.models.self_supervised.cpc.networks`),
 270
`interpolate_latent_space()`
 (`pl_bolts.callbacks.LatentDimInterpolator`
 method), 181
`interpolate_latent_space()`
 (`pl_bolts.callbacks.variational.LatentDimInterpolator`
 method), 196
`iou()` (in module `pl_bolts.metrics.object_detection`),
 257
`iou_loss()` (in module `pl_bolts.losses.object_detection`), 383

K

`KittiDataModule` (class in `pl_bolts.datamodules`),
 208
`KittiDataModule` (class in `pl_bolts.datamodules.kitti_datamodule`),
 231
`KittiDataset` (class in `pl_bolts.datamodules`), 216
`KittiDataset` (class in `pl_bolts.datasets`), 247
`KittiDataset` (class in `pl_bolts.datasets.kitti_dataset`), 255
`KNNOnlineEvaluator` (class in `pl_bolts.callbacks.knn_online`), 193

L

`labels` (`pl_bolts.datasets.CIFAR10` attribute), 244
`labels` (`pl_bolts.datasets.cifar10_dataset.CIFAR10` at-
 tribute), 250
`LARSWrapper` (class in `pl_bolts.optimizers`), 387
`LARSWrapper` (class in `pl_bolts.optimizers.lars_scheduling`), 388
`LatentDimInterpolator` (class in `pl_bolts.callbacks`), 181
`load_instance()` (`pl_bolts.datamodules.async_dataloader.Asynchronous`
 method), 217
`load_instance()` (`pl_bolts.datamodules.AsynchronousLoader`
 method), 197
`load_loop()` (`pl_bolts.datamodules.async_dataloader.Asynchronous`
 method), 217
`load_loop()` (`pl_bolts.datamodules.AsynchronousLoader`
 method), 197
`load_pretrained()`
 (`pl_bolts.models.self_supervised.cpc.cpc_module.CPCV2`
 method), 350
`load_pretrained()`
 (`pl_bolts.models.self_supervised.cpc.CPCV2`
 method), 345
`load_pretrained()`
 (`pl_bolts.models.self_supervised.CPCV2`
 method), 326
`log_histogram()` (`pl_bolts.callbacks.data_monitor.DataMonitorBase`
 method), 190
`log_histograms()` (`pl_bolts.callbacks.data_monitor.DataMonitorBase`
 method), 190
`LogisticRegression` (class in `pl_bolts.models`),
 260
`LogisticRegression` (class in `pl_bolts.models.regression`), 284
`LogisticRegression` (class in `pl_bolts.models.regression.logistic_regression`),
 286
`loss()` (`pl_bolts.models.rl.Reinforce` method), 296
`loss()` (`pl_bolts.models.rl.reinforce_model.Reinforce`
 method), 319
`loss()` (`pl_bolts.models.rl.vanilla_policy_gradient_model.VanillaPolicyG`
 method), 321
`loss()` (`pl_bolts.models.rl.VanillaPolicyGradient`

- method*), 299
- ## M
- `make_environment()` (in module `pl_bolts.models.rl.common.gym_wrappers`), 302
- `make_environment()` (`pl_bolts.models.rl.DQN` static method), 289
- `make_environment()` (`pl_bolts.models.rl.dqn_model.DQN` static method), 311
- `MASK_PATH` (`pl_bolts.datamodules.KittiDataset` attribute), 217
- `MASK_PATH` (`pl_bolts.datasets.kitti_dataset.KittiDataset` attribute), 255
- `MASK_PATH` (`pl_bolts.datasets.KittiDataset` attribute), 248
- `MaxAndSkipEnv` (class in `pl_bolts.models.rl.common.gym_wrappers`), 301
- `MaybeBatchNorm2d` (class in `pl_bolts.models.self_supervised.amdim.networks`), 340
- `mean()` (in module `pl_bolts.metrics`), 256
- `mean()` (in module `pl_bolts.metrics.aggregation`), 256
- `mean()` (`pl_bolts.models.rl.common.memory.MeanBuffer` method), 303
- `MeanBuffer` (class in `pl_bolts.models.rl.common.memory`), 303
- `message()` (`pl_bolts.callbacks.BatchGradientVerificationCallback` attribute), 198
- `message()` (`pl_bolts.callbacks.batch_verification.base.VerificationCallback` attribute), 184
- `message()` (`pl_bolts.callbacks.batch_verification.batch_gradient.VerificationCallback` attribute), 186
- `MLP` (class in `pl_bolts.models.rl.common.networks`), 306
- `MLP` (class in `pl_bolts.models.self_supervised.byol.models`), 344
- `MLP` (class in `pl_bolts.models.self_supervised.simsiam.models`), 361
- `MNISTDataModule` (class in `pl_bolts.datamodules`), 209
- `MNISTDataModule` (class in `pl_bolts.datamodules.mnist_datamodule`), 233
- `Moco2EvalCIFAR10Transforms` (class in `pl_bolts.models.self_supervised.moco.transforms`), 357
- `Moco2EvalImagenetTransforms` (class in `pl_bolts.models.self_supervised.moco.transforms`), 357
- `Moco2EvalSTL10Transforms` (class in `pl_bolts.models.self_supervised.moco.transforms`), 357
- `Moco2TrainCIFAR10Transforms` (class in `pl_bolts.models.self_supervised.moco.transforms`), 357
- `Moco2TrainImagenetTransforms` (class in `pl_bolts.models.self_supervised.moco.transforms`), 357
- `Moco2TrainSTL10Transforms` (class in `pl_bolts.models.self_supervised.moco.transforms`), 357
- `MocoLRScheduler` (class in `pl_bolts.models.self_supervised.moco.callbacks`), 354
- `MocoV2` (class in `pl_bolts.models.self_supervised`), 326
- `MocoV2` (class in `pl_bolts.models.self_supervised.moco.moco2_module`), 355
- `ModuleDataMonitor` (class in `pl_bolts.callbacks`), 178
- `ModuleDataMonitor` (class in `pl_bolts.callbacks.data_monitor`), 190
- `MultiPrototypes` (class in `pl_bolts.models.self_supervised.swav.swav_resnet`), 369
- `MultiStepBuffer` (class in `pl_bolts.models.rl.common.memory`), 303
- ## N
- `name` (`pl_bolts.datamodules.binary_mnist_datamodule.BinaryMNISTDataModule` attribute), 219
- `name` (`pl_bolts.datamodules.BinaryMNISTDataModule` attribute), 219
- `name` (`pl_bolts.datamodules.cifar10_datamodule.CIFAR10DataModule` attribute), 221
- `name` (`pl_bolts.datamodules.CIFAR10DataModule` attribute), 221
- `name` (`pl_bolts.datamodules.cityscapes_datamodule.CityscapesDataModule` attribute), 224
- `name` (`pl_bolts.datamodules.CityscapesDataModule` attribute), 202
- `name` (`pl_bolts.datamodules.fashion_mnist_datamodule.FashionMNISTDataModule` attribute), 228
- `name` (`pl_bolts.datamodules.FashionMNISTDataModule` attribute), 206
- `name` (`pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule` attribute), 231
- `name` (`pl_bolts.datamodules.ImagenetDataModule` attribute), 208
- `name` (`pl_bolts.datamodules.kitti_datamodule.KittiDataModule` attribute), 232
- `name` (`pl_bolts.datamodules.KittiDataModule` attribute), 209
- `name` (`pl_bolts.datamodules.mnist_datamodule.MNISTDataModule` attribute), 234
- `name` (`pl_bolts.datamodules.MNISTDataModule` attribute), 211

<code>on_after_backward()</code> (<i>pl_bolts.models.self_supervised.swav.SwAV</i> method), 365	(<i>pl_bolts.callbacks.data_monitor.TrainingDataMonitor</i> method), 192
<code>on_after_backward()</code> (<i>pl_bolts.models.self_supervised.swav.swav_module.SwAV</i> method), 368	<code>on_train_batch_start()</code> (<i>pl_bolts.callbacks.TrainingDataMonitor</i> method), 179
<code>on_epoch_end()</code> (<i>pl_bolts.callbacks.LatentDimInterpolator</i> method), 181	<code>on_train_end()</code> (<i>pl_bolts.callbacks.data_monitor.ModuleDataMonitor</i> method), 191
<code>on_epoch_end()</code> (<i>pl_bolts.callbacks.printing.PrintTableMetricsCallback</i> method), 194	<code>on_train_end()</code> (<i>pl_bolts.callbacks.ModuleDataMonitor</i> method), 179
<code>on_epoch_end()</code> (<i>pl_bolts.callbacks.PrintTableMetricsCallback</i> method), 180	<code>on_train_epoch_end()</code> (<i>pl_bolts.models.self_supervised.SwAV</i> method), 332
<code>on_epoch_end()</code> (<i>pl_bolts.callbacks.TensorboardGenerativeModelImageSampler</i> method), 183	<code>on_train_epoch_end()</code> (<i>pl_bolts.models.self_supervised.swav.SwAV</i> method), 365
<code>on_epoch_end()</code> (<i>pl_bolts.callbacks.variational.LatentDimInterpolator</i> method), 196	<code>on_train_epoch_end()</code> (<i>pl_bolts.callbacks.variational.LatentDimInterpolator</i> method), 368
<code>on_epoch_end()</code> (<i>pl_bolts.callbacks.vision.image_generation.TensorboardGenerativeModelImageSampler</i> method), 189	<code>on_train_epoch_start()</code> (<i>pl_bolts.callbacks.data_monitor.TrainingDataMonitor</i> method), 179
<code>on_epoch_start()</code> (<i>pl_bolts.models.self_supervised.moco.callbacks.MocoLRScheduler</i> method), 354	<code>on_train_epoch_start()</code> (<i>pl_bolts.models.self_supervised.ssl_finetuner.SSLLFineTuner</i> method), 375
<code>on_pretrain_routine_start()</code> (<i>pl_bolts.callbacks.ssl_online.SSLOnlineEvaluator</i> method), 195	<code>on_train_epoch_start()</code> (<i>pl_bolts.models.self_supervised.SSLLFineTuner</i> method), 331
<code>on_pretrain_routine_start()</code> (<i>pl_bolts.callbacks.SSLOnlineEvaluator</i> method), 180	<code>on_train_epoch_start()</code> (<i>pl_bolts.models.self_supervised.SwAV</i> method), 332
<code>on_train_batch_end()</code> (<i>pl_bolts.callbacks.byol_updates.BYOLMAWeightUpdate</i> method), 189	<code>on_train_epoch_start()</code> (<i>pl_bolts.models.self_supervised.swav.SwAV</i> method), 365
<code>on_train_batch_end()</code> (<i>pl_bolts.callbacks.BYOLMAWeightUpdate</i> method), 178	<code>on_train_epoch_start()</code> (<i>pl_bolts.models.self_supervised.swav.swav_module.SwAV</i> method), 368
<code>on_train_batch_end()</code> (<i>pl_bolts.callbacks.ConfusedLogitCallback</i> method), 182	<code>on_train_epoch_start()</code> (<i>pl_bolts.callbacks.data_monitor.TrainingDataMonitor</i> method), 179
<code>on_train_batch_end()</code> (<i>pl_bolts.callbacks.ssl_online.SSLOnlineEvaluator</i> method), 195	<code>on_train_start()</code> (<i>pl_bolts.callbacks.BatchGradientVerificationCallback</i> method), 178
<code>on_train_batch_end()</code> (<i>pl_bolts.callbacks.SSLOnlineEvaluator</i> method), 181	<code>on_train_start()</code> (<i>pl_bolts.callbacks.data_monitor.DataMonitorBase</i> method), 190
<code>on_train_batch_end()</code> (<i>pl_bolts.callbacks.vision.confused_logit.ConfusedLogitCallback</i> method), 188	<code>on_train_start()</code> (<i>pl_bolts.callbacks.data_monitor.ModuleDataMonitor</i> method), 191
<code>on_train_batch_end()</code> (<i>pl_bolts.models.self_supervised.BYOL</i> method), 325	<code>on_train_start()</code> (<i>pl_bolts.callbacks.ModuleDataMonitor</i> method), 179
<code>on_train_batch_end()</code> (<i>pl_bolts.models.self_supervised.byol.byol_module.BYOL</i> method), 344	<code>on_train_start()</code> (<i>pl_bolts.callbacks.validation.batch_gradient.BatchGradientVerificationCallback</i> method), 186
<code>on_train_batch_start()</code> (<i>pl_bolts.callbacks.data_monitor.DataMonitorBase</i> method), 190	<code>on_train_start()</code> (<i>pl_bolts.models.rl.noisy_dqn_model.NoisyDQN</i> method), 315
<code>on_train_batch_start()</code> (<i>pl_bolts.callbacks.data_monitor.TrainingDataMonitor</i> method), 179	<code>on_train_start()</code> (<i>pl_bolts.models.rl.NoisyDQN</i> method), 293
<code>on_validation_batch_end()</code> (<i>pl_bolts.callbacks.ssl_online.SSLOnlineEvaluator</i> method), 195	<code>on_validation_batch_end()</code> (<i>pl_bolts.callbacks.SSLOnlineEvaluator</i> method), 181
<code>on_validation_batch_end()</code> (<i>pl_bolts.callbacks.SSLOnlineEvaluator</i> method), 181	<code>on_validation_epoch_end()</code>

`(pl_bolts.callbacks.knn_online.KNNOnlineEvaluator`
`method), 193`
`optimizer_step()` (`pl_bolts.models.self_supervised.SimCLR`
`method), 328`
`optimizer_step()` (`pl_bolts.models.self_supervised.simclr.simclr`
`method), 359`
`optimizer_step()` (`pl_bolts.models.self_supervised.SimSiam`
`method), 330`
`optimizer_step()` (`pl_bolts.models.self_supervised.simsiam.simsiam`
`method), 363`
`optimizer_step()` (`pl_bolts.models.self_supervised.SwAV`
`method), 333`
`optimizer_step()` (`pl_bolts.models.self_supervised.swav.swav`
`method), 365`
`optimizer_step()` (`pl_bolts.models.self_supervised.swav.swav`
`method), 369`

P
`param_groups()` (`pl_bolts.optimizers.lars_scheduling.LARSWrapper`
`property), 389`
`param_groups()` (`pl_bolts.optimizers.LARSWrapper`
`property), 387`
`parse_devkit_archive()` (in module
`pl_bolts.datasets), 248`
`parse_devkit_archive()` (in module
`pl_bolts.datasets.imagenet_dataset), 254`
`parse_map_indexes()`
`(pl_bolts.losses.self_supervised_learning.FeatureMapContrastiveTask`
`static method), 386`
`partition_train_set()`
`(pl_bolts.datasets.imagenet_dataset.UnlabeledImagenet`
`method), 254`
`partition_train_set()`
`(pl_bolts.datasets.UnlabeledImagenet method),`
`247`
`Patchify` (class in `pl_bolts.transforms.self_supervised.ssl_transforms`),
`390`
`per_dqn_loss()` (in module `pl_bolts.losses.rl`), 384
`PERBuffer` (class in
`pl_bolts.models.rl.common.memory), 304`
`PERDQN` (class in `pl_bolts.models.rl`), 293
`PERDQN` (class in `pl_bolts.models.rl.per_dqn_model`),
`316`
`PixelCNN` (class in `pl_bolts.models`), 261
`PixelCNN` (class in `pl_bolts.models.vision`), 375
`PixelCNN` (class in `pl_bolts.models.vision.pixel_cnn`),
`380`
`pl_bolts.callbacks` (module), 177
`pl_bolts.callbacks.byol_updates` (module),
`189`
`pl_bolts.callbacks.data_monitor` (module),
`190`
`pl_bolts.callbacks.knn_online` (module),
`193`
`pl_bolts.callbacks.printing` (module), 193
`pl_bolts.callbacks.ssl_online` (module),
`196`
`pl_bolts.callbacks.variational` (module),
`196`
`pl_bolts.callbacks.verifiacation` (module),
`196`
`pl_bolts.callbacks.verifiacation.base`
`pl_bolts.callbacks.verifiacation.batch_gradient`
`pl_bolts.callbacks.vision` (module), 187
`pl_bolts.callbacks.vision.confused_logit`
`(module), 187`
`pl_bolts.callbacks.vision.image_generation`
`(module), 188`
`pl_bolts.datamodules` (module), 196
`pl_bolts.datamodules.async_dataloader`
`(module), 217`
`pl_bolts.datamodules.binary_mnist_datamodule`
`(module), 218`
`pl_bolts.datamodules.cifar10_datamodule`
`(module), 219`
`pl_bolts.datamodules.cityscapes_datamodule`
`(module), 222`
`pl_bolts.datamodules.experience_source`
`(module), 224`
`pl_bolts.datamodules.fashion_mnist_datamodule`
`(module), 227`
`pl_bolts.datamodules.imagenet_datamodule`
`(module), 229`
`pl_bolts.datamodules.kitti_datamodule`
`(module), 231`
`pl_bolts.datamodules.mnist_datamodule`
`(module), 233`
`pl_bolts.datamodules.sklearn_datamodule`
`(module), 234`
`pl_bolts.datamodules.ssl_imagenet_datamodule`
`(module), 237`
`pl_bolts.datamodules.stl10_datamodule`
`(module), 237`
`pl_bolts.datamodules.vision_datamodule`
`(module), 240`
`pl_bolts.datamodules.vocdetection_datamodule`
`(module), 241`
`pl_bolts.datasets` (module), 242
`pl_bolts.datasets.base_dataset` (module),
`249`
`pl_bolts.datasets.cifar10_dataset` (mod-
`ule), 249`
`pl_bolts.datasets.concat_dataset` (mod-
`ule), 251`
`pl_bolts.datasets.dummy_dataset` (module),
`252`

[pl_bolts.datasets.imagenet_dataset \(module\), 254](#)
[pl_bolts.datasets.kitti_dataset \(module\), 255](#)
[pl_bolts.datasets.mnist_dataset \(module\), 255](#)
[pl_bolts.datasets.ssl_amdim_datasets \(module\), 256](#)
[pl_bolts.losses \(module\), 382](#)
[pl_bolts.losses.object_detection \(module\), 382](#)
[pl_bolts.losses.rl \(module\), 384](#)
[pl_bolts.losses.self_supervised_learning \(module\), 385](#)
[pl_bolts.metrics \(module\), 256](#)
[pl_bolts.metrics.aggregation \(module\), 256](#)
[pl_bolts.metrics.object_detection \(module\), 256](#)
[pl_bolts.models \(module\), 257](#)
[pl_bolts.models.autoencoders \(module\), 265](#)
[pl_bolts.models.autoencoders.basic_ae \(module\), 267](#)
[pl_bolts.models.autoencoders.basic_ae.basic_ae \(module\), 267](#)
[pl_bolts.models.autoencoders.basic_vae \(module\), 268](#)
[pl_bolts.models.autoencoders.basic_vae.basic_vae \(module\), 269](#)
[pl_bolts.models.autoencoders.components \(module\), 270](#)
[pl_bolts.models.detection \(module\), 272](#)
[pl_bolts.models.detection.components \(module\), 273](#)
[pl_bolts.models.detection.components._supported_models \(module\), 273](#)
[pl_bolts.models.detection.components.torchvision_modules \(module\), 273](#)
[pl_bolts.models.detection.faster_rcnn \(module\), 274](#)
[pl_bolts.models.detection.faster_rcnn.backbones \(module\), 275](#)
[pl_bolts.models.detection.faster_rcnn.faster_rcnn \(module\), 276](#)
[pl_bolts.models.gans \(module\), 277](#)
[pl_bolts.models.gans.basic \(module\), 279](#)
[pl_bolts.models.gans.basic.basic_gan_module \(module\), 279](#)
[pl_bolts.models.gans.basic.components \(module\), 281](#)
[pl_bolts.models.gans.dcgan \(module\), 281](#)
[pl_bolts.models.gans.dcgan.components \(module\), 281](#)
[pl_bolts.models.gans.dcgan.dcgan_module \(module\), 282](#)
[pl_bolts.models.mnist_module \(module\), 382](#)
[pl_bolts.models.regression \(module\), 283](#)
[pl_bolts.models.regression.linear_regression \(module\), 285](#)
[pl_bolts.models.regression.logistic_regression \(module\), 286](#)
[pl_bolts.models.rl \(module\), 286](#)
[pl_bolts.models.rl.common \(module\), 299](#)
[pl_bolts.models.rl.common.agents \(module\), 299](#)
[pl_bolts.models.rl.common.cli \(module\), 301](#)
[pl_bolts.models.rl.common.gym_wrappers \(module\), 301](#)
[pl_bolts.models.rl.common.memory \(module\), 302](#)
[pl_bolts.models.rl.common.networks \(module\), 305](#)
[pl_bolts.models.rl.double_dqn_model \(module\), 308](#)
[pl_bolts.models.rl.dqn_model \(module\), 310](#)
[pl_bolts.models.rl.dueling_dqn_model \(module\), 310](#)
[pl_bolts.models.rl.noisy_dqn_model \(module\), 314](#)
[pl_bolts.models.rl.per_dqn_model \(module\), 315](#)
[pl_bolts.models.rl.reinforce_model \(module\), 318](#)
[pl_bolts.models.rl.vanilla_policy_gradient_model \(module\), 320](#)
[pl_bolts.models.self_supervised \(module\), 322](#)
[pl_bolts.models.self_supervised.amdim \(module\), 333](#)
[pl_bolts.models.self_supervised.amdim.amdim_module \(module\), 337](#)
[pl_bolts.models.self_supervised.amdim.datasets \(module\), 339](#)
[pl_bolts.models.self_supervised.amdim.networks \(module\), 339](#)
[pl_bolts.models.self_supervised.amdim.transforms \(module\), 340](#)
[pl_bolts.models.self_supervised.byol \(module\), 343](#)
[pl_bolts.models.self_supervised.byol.byol_module \(module\), 343](#)
[pl_bolts.models.self_supervised.byol.models \(module\), 344](#)
[pl_bolts.models.self_supervised.cpc \(module\), 345](#)
[pl_bolts.models.self_supervised.cpc.cpc_finetuner \(module\), 349](#)
[pl_bolts.models.self_supervised.cpc.cpc_module \(module\), 349](#)

- method), 250
- prepare_data() (*pl_bolts.datasets.cifar10_dataset.TrialCIFAR10* attribute), 251
- prepare_data() (*pl_bolts.datasets.TrialCIFAR10* attribute), 245
- prepare_data() (*pl_bolts.models.LitMNIST* attribute), 259
- prepare_data() (*pl_bolts.models.mnist_module.LitMNIST* attribute), 382
- pretrained_urls (*pl_bolts.models.AE* attribute), 258
- pretrained_urls (*pl_bolts.models.autoencoders.AE* attribute), 265
- pretrained_urls (*pl_bolts.models.autoencoders.basic_ae.basic_ae_module.AE* attribute), 268
- pretrained_urls (*pl_bolts.models.autoencoders.basic_vae.basic_vae_module.VAE* attribute), 270
- pretrained_urls (*pl_bolts.models.autoencoders.VAE* attribute), 266
- pretrained_urls (*pl_bolts.models.VAE* attribute), 259
- pretrained_weights_available() (*pl_bolts.models.AE* static method), 258
- pretrained_weights_available() (*pl_bolts.models.autoencoders.AE* static method), 265
- pretrained_weights_available() (*pl_bolts.models.autoencoders.basic_ae.basic_ae_module.AE* static method), 268
- pretrained_weights_available() (*pl_bolts.models.autoencoders.basic_vae.basic_vae_module.VAE* static method), 269
- pretrained_weights_available() (*pl_bolts.models.autoencoders.VAE* static method), 266
- pretrained_weights_available() (*pl_bolts.models.VAE* static method), 259
- PrintTableMetricsCallback (class in *pl_bolts.callbacks*), 179
- PrintTableMetricsCallback (class in *pl_bolts.callbacks.printing*), 193
- process() (*pl_bolts.models.rl.common.gym_wrappers.ProcessFrame84* static method), 302
- ProcessFrame84 (class in *pl_bolts.models.rl.common.gym_wrappers*), 302
- Projection (class in *pl_bolts.models.self_supervised.simclr.simclr_module*), 358
- RandomDictDataset (class in *pl_bolts.datasets*), 246
- RandomDictDataset (class in *pl_bolts.datasets.dummy_dataset*), 252
- RandomDictDataset (class in *pl_bolts.datasets.dummy_dataset*), 253
- RandomDictStringDataset (class in *pl_bolts.datasets*), 246
- RandomDictStringDataset (class in *pl_bolts.datasets.dummy_dataset*), 253
- RandomTranslateWithReflect (class in *pl_bolts.transforms.self_supervised.ssl_transforms*), 390
- Reinforce (class in *pl_bolts.models.rl*), 295
- Reinforce (class in *pl_bolts.models.rl.reinforce_model*), 318
- relabel (*pl_bolts.datasets.cifar10_dataset.CIFAR10* attribute), 244
- ReplayBuffer (class in *pl_bolts.models.rl.common.memory*), 305
- reset() (*pl_bolts.models.rl.common.gym_wrappers.BufferWrapper* method), 301
- reset() (*pl_bolts.models.rl.common.gym_wrappers.FireResetEnv* method), 301
- reset() (*pl_bolts.models.rl.common.gym_wrappers.MaxAndSkipEnv* method), 302
- reset() (*pl_bolts.models.rl.common.gym_wrappers.ToTensor* method), 302
- reset_parameters() (*pl_bolts.models.rl.common.networks.NoisyLinear* method), 308
- resize_conv1x1() (in *pl_bolts.models.autoencoders.components*), 271
- resize_conv3x3() (in *pl_bolts.models.autoencoders.components*), 271
- ResNet (class in *pl_bolts.models.self_supervised.resnets*), 372
- ResNet (class in *pl_bolts.models.self_supervised.swav.swav_resnet*), 369
- resnet101() (in *pl_bolts.models.self_supervised.resnets*), 374
- resnet152() (in *pl_bolts.models.self_supervised.resnets*), 373
- resnet18() (in *pl_bolts.models.self_supervised.resnets*), 372
- resnet18() (in *pl_bolts.models.self_supervised.swav*), 366
- resnet18() (in *pl_bolts.models.self_supervised.swav.swav_resnet*), 370
- resnet18_decoder() (in *pl_bolts.models.self_supervised.swav.swav_resnet*), 370

R

[pl_bolts.models.autoencoders](#)), 266
[resnet18_decoder\(\)](#) (in module [pl_bolts.models.autoencoders.components](#)), 271
[resnet18_encoder\(\)](#) (in module [pl_bolts.models.autoencoders](#)), 266
[resnet18_encoder\(\)](#) (in module [pl_bolts.models.autoencoders.components](#)), 271
[resnet34\(\)](#) (in module [pl_bolts.models.self_supervised.resnets](#)), 372
[resnet50\(\)](#) (in module [pl_bolts.models.self_supervised.resnets](#)), 372
[resnet50\(\)](#) (in module [pl_bolts.models.self_supervised.swav](#)), 366
[resnet50\(\)](#) (in module [pl_bolts.models.self_supervised.swav.swav_resnet](#)), 370
[resnet50_decoder\(\)](#) (in module [pl_bolts.models.autoencoders](#)), 266
[resnet50_decoder\(\)](#) (in module [pl_bolts.models.autoencoders.components](#)), 271
[resnet50_encoder\(\)](#) (in module [pl_bolts.models.autoencoders](#)), 266
[resnet50_encoder\(\)](#) (in module [pl_bolts.models.autoencoders.components](#)), 271
[resnet50w2\(\)](#) (in module [pl_bolts.models.self_supervised.swav.swav_resnet](#)), 370
[resnet50w4\(\)](#) (in module [pl_bolts.models.self_supervised.swav.swav_resnet](#)), 370
[resnet50w5\(\)](#) (in module [pl_bolts.models.self_supervised.swav.swav_resnet](#)), 370
[ResNetDecoder](#) (class in [pl_bolts.models.autoencoders.components](#)), 271
[ResNetEncoder](#) (class in [pl_bolts.models.autoencoders.components](#)), 271
[resnext101_32x8d\(\)](#) (in module [pl_bolts.models.self_supervised.resnets](#)), 373
[resnext50_32x4d\(\)](#) (in module [pl_bolts.models.self_supervised.resnets](#)), 373
[reward\(\)](#) ([pl_bolts.datamodules.experience_source.Experience](#) property), 225
[reward\(\)](#) ([pl_bolts.models.rl.common.memory.Experience](#) property), 303
[run_cli\(\)](#) (in module [pl_bolts.models.detection.faster_rcnn.faster_rcnn_module](#)), 277
[run_n_episodes\(\)](#) ([pl_bolts.models.rl.DQN](#) method), 290
[run_n_episodes\(\)](#) ([pl_bolts.models.rl.dqn_model.DQN](#) method), 312
[runner\(\)](#) ([pl_bolts.datamodules.DiscountedExperienceSource](#) method), 203
[runner\(\)](#) ([pl_bolts.datamodules.experience_source.BaseExperienceSource](#) method), 224
[runner\(\)](#) ([pl_bolts.datamodules.experience_source.DiscountedExperienceSource](#) method), 225
[runner\(\)](#) ([pl_bolts.datamodules.experience_source.ExperienceSource](#) method), 226
[runner\(\)](#) ([pl_bolts.datamodules.ExperienceSource](#) method), 204
S
[sample\(\)](#) ([pl_bolts.models.autoencoders.basic_vae.basic_vae_module.VAE](#) method), 269
[sample\(\)](#) ([pl_bolts.models.autoencoders.VAE](#) method), 266
[sample\(\)](#) ([pl_bolts.models.rl.common.memory.Buffer](#) method), 302
[sample\(\)](#) ([pl_bolts.models.rl.common.memory.PERBuffer](#) method), 304
[sample\(\)](#) ([pl_bolts.models.rl.common.memory.ReplayBuffer](#) method), 305
[sample\(\)](#) ([pl_bolts.models.VAE](#) method), 259
[ScaledFloatFrame](#) (class in [pl_bolts.models.rl.common.gym_wrappers](#)), 302
[select_nb_imgs_per_class\(\)](#) ([pl_bolts.datasets.ssl_anddim_datasets.SSLDatasetMixin](#) class method), 256
[select_nb_imgs_per_class\(\)](#) ([pl_bolts.datasets.SSLDatasetMixin](#) class method), 248
[SemSegment](#) (class in [pl_bolts.models](#)), 261
[SemSegment](#) (class in [pl_bolts.models.vision](#)), 375
[SemSegment](#) (class in [pl_bolts.models.vision.segmentation](#)), 380
[setup\(\)](#) ([pl_bolts.datamodules.vision_datamodule.VisionDataModule](#) method), 241
[setup\(\)](#) ([pl_bolts.models.self_supervised.SwAV](#) method), 333
[setup\(\)](#) ([pl_bolts.models.self_supervised.swav.SwAV](#) method), 365
[setup\(\)](#) ([pl_bolts.models.self_supervised.swav.swav_module.SwAV](#) method), 369
[shape2str\(\)](#) (in module [pl_bolts.callbacks.data_monitor](#)), 192

[shared_step\(\)](#) (*pl_bolts.models.self_supervised.BYOL* [method](#)), 325
[shared_step\(\)](#) (*pl_bolts.models.self_supervised.byol.byol_module.BYOL* [method](#)), 344
[shared_step\(\)](#) (*pl_bolts.models.self_supervised.cpc.cpc_module.CPCV2* [method](#)), 350
[shared_step\(\)](#) (*pl_bolts.models.self_supervised.cpc.CPCV2* [method](#)), 345
[shared_step\(\)](#) (*pl_bolts.models.self_supervised.CPCV2* [method](#)), 326
[shared_step\(\)](#) (*pl_bolts.models.self_supervised.SimCLR* [method](#)), 328
[shared_step\(\)](#) (*pl_bolts.models.self_supervised.simclr.simclr_module.SimCLR* [method](#)), 359
[shared_step\(\)](#) (*pl_bolts.models.self_supervised.ssl_finetuner.SSLFineTuner* [method](#)), 375
[shared_step\(\)](#) (*pl_bolts.models.self_supervised.SSLFineTuner* [method](#)), 331
[shared_step\(\)](#) (*pl_bolts.models.self_supervised.SwAV* [method](#)), 333
[shared_step\(\)](#) (*pl_bolts.models.self_supervised.swav.SwAV* [method](#)), 365
[shared_step\(\)](#) (*pl_bolts.models.self_supervised.swav.swav_module.SwAV* [method](#)), 369
[SiameseArm](#) (class in *pl_bolts.models.self_supervised*), 330
[SiameseArm](#) (class in *pl_bolts.models.self_supervised.byol.models*), 344
[SiameseArm](#) (class in *pl_bolts.models.self_supervised.simsiam.models*), 361
[SimCLR](#) (class in *pl_bolts.models.self_supervised*), 327
[SimCLR](#) (class in *pl_bolts.models.self_supervised.simclr.simclr_module*), 358
[SimCLREvalDataTransform](#) (class in *pl_bolts.models.self_supervised.simclr.transforms*), 359
[SimCLRFinetuneTransform](#) (class in *pl_bolts.models.self_supervised.simclr.transforms*), 360
[SimCLRTrainDataTransform](#) (class in *pl_bolts.models.self_supervised.simclr.transforms*), 360
[SimSiam](#) (class in *pl_bolts.models.self_supervised*), 328
[SimSiam](#) (class in *pl_bolts.models.self_supervised.simsiam.models*), 362
[sinkhorn\(\)](#) (*pl_bolts.models.self_supervised.SwAV* [method](#)), 333
[sinkhorn\(\)](#) (*pl_bolts.models.self_supervised.swav.SwAV* [method](#)), 365
[sinkhorn\(\)](#) (*pl_bolts.models.self_supervised.swav.swav_module.SwAV* [method](#)), 369
[SklearnDataModule](#) (class in *pl_bolts.datamodules*), 211
[SklearnDataModule](#) (class in *pl_bolts.datamodules.sklearn_datamodule*), 234
[SklearnDataset](#) (class in *pl_bolts.datamodules*), 212
[SklearnDataset](#) (class in *pl_bolts.datamodules.sklearn_datamodule*), 235
[split_head_tail_exp\(\)](#) (*pl_bolts.datamodules.DiscountedExperienceSource* [method](#)), 203
[split_head_tail_exp\(\)](#) (*pl_bolts.datamodules.experience_source.DiscountedExperienceSource* [method](#)), 225
[split_head_tail_exp\(\)](#) (*pl_bolts.models.rl.common.memory.MultiStepBuffer* [method](#)), 304
[SSLDatasetMixin](#) (class in *pl_bolts.datasets*), 248
[SSLDatasetMixin](#) (class in *pl_bolts.datasets.ssl_amdim_datasets*), 256
[SSLEvaluator](#) (class in *pl_bolts.models.self_supervised*), 326
[SSLEvaluator](#) (class in *pl_bolts.models.self_supervised.evaluator*), 327
[SSLFineTuner](#) (class in *pl_bolts.models.self_supervised*), 330
[SSLFineTuner](#) (class in *pl_bolts.models.self_supervised.ssl_finetuner*), 374
[SSLImagenetDataModule](#) (class in *pl_bolts.datamodules*), 212
[SSLImagenetDataModule](#) (class in *pl_bolts.datamodules.ssl_imagenet_datamodule*), 237
[SSLOnlineEvaluator](#) (class in *pl_bolts.callbacks*), 180
[SSLOnlineEvaluator](#) (class in *pl_bolts.callbacks.ssl_online*), 195
[state\(\)](#) (*pl_bolts.datamodules.experience_source.Experience* [property](#)), 225
[state\(\)](#) (*pl_bolts.models.rl.common.memory.Experience* [property](#)), 303
[state\(\)](#) (*pl_bolts.optimizers.lars_scheduling.LARSWrapper* [property](#)), 389
[state\(\)](#) (*pl_bolts.optimizers.LARSWrapper* [property](#)), 387
[step\(\)](#) (*pl_bolts.models.AE* [method](#)), 258
[step\(\)](#) (*pl_bolts.models.autoencoders.AE* [method](#)), 265
[step\(\)](#) (*pl_bolts.models.autoencoders.basic_ae.basic_ae_module.AE* [method](#)), 268
[step\(\)](#) (*pl_bolts.models.autoencoders.basic_vae.basic_vae_module.VAE* [method](#)), 270
[step\(\)](#) (*pl_bolts.models.autoencoders.VAE* [method](#)), 266
[step\(\)](#) (*pl_bolts.models.rl.common.gym_wrappers.FireResetEnv* [method](#)), 389

`method`), 301
`step()` (`pl_bolts.models.rl.common.gym_wrappers.MaxAndSkipEnv` attribute), 249
`method`), 302
`step()` (`pl_bolts.models.rl.common.gym_wrappers.ToTensor` attribute), 250
`method`), 302
`step()` (`pl_bolts.models.VAE` method), 259
`step()` (`pl_bolts.optimizers.lars_scheduling.LARSWrapper` attribute), 251
`method`), 389
`step()` (`pl_bolts.optimizers.LARSWrapper` method), 245
`387`
`stl()` (`pl_bolts.models.self_supervised.amdim.datasets.AMDHPretrainingGenerativeModelImageSampler` static method), 339
`stl()` (`pl_bolts.models.self_supervised.amdim.datasets.AMDHPretrainingGenerativeModelImageSampler` static method), 339
`stl10_normalization()` (in module `pl_bolts.transforms.dataset_normalizations`), 391
`STL10DataModule` (class in `pl_bolts.datamodules`), 213
`STL10DataModule` (class in `pl_bolts.datamodules.stl10_datamodule`), 237
`supported_loggers` (`pl_bolts.callbacks.data_monitor.DataMonitorBase` attribute), 190
`SwAV` (class in `pl_bolts.models.self_supervised`), 331
`SwAV` (class in `pl_bolts.models.self_supervised.swav`), 364
`SwAV` (class in `pl_bolts.models.self_supervised.swav.swav_module`), 367
`SwAVEvalDataTransform` (class in `pl_bolts.models.self_supervised.swav`), 365
`SwAVEvalDataTransform` (class in `pl_bolts.models.self_supervised.swav.transforms`), 370
`SwAVFinetuneTransform` (class in `pl_bolts.models.self_supervised.swav`), 366
`SwAVFinetuneTransform` (class in `pl_bolts.models.self_supervised.swav.transforms`), 371
`SwAVTrainDataTransform` (class in `pl_bolts.models.self_supervised.swav`), 366
`SwAVTrainDataTransform` (class in `pl_bolts.models.self_supervised.swav.transforms`), 371
`SyncFunction` (class in `pl_bolts.models.self_supervised.simclr.simclr_module`), 359
T
`tanh_clip()` (in module `pl_bolts.losses.self_supervised_learning`), 386
`targets` (`pl_bolts.datasets.base_dataset.LightDataset` attribute), 244
`targets` (`pl_bolts.datasets.CIFAR10` attribute), 244
`targets` (`pl_bolts.datasets.cifar10_dataset.CIFAR10` attribute), 250
`targets` (`pl_bolts.datasets.cifar10_dataset.TrialCIFAR10` attribute), 251
`targets` (`pl_bolts.datasets.LightDataset` attribute), 243
`targets` (`pl_bolts.datasets.TrialCIFAR10` attribute), 245
`TensorDataset` (class in `pl_bolts.datamodules`), 212
`TensorDataset` (class in `pl_bolts.datamodules.sklearn_datamodule`), 236
`test_dataloader()` (`pl_bolts.datamodules.cityscapes_datamodule.CityscapesDataModule` method), 224
`test_dataloader()` (`pl_bolts.datamodules.CityscapesDataModule` method), 202
`test_dataloader()` (`pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule` method), 230
`test_dataloader()` (`pl_bolts.datamodules.ImagenetDataModule` method), 207
`test_dataloader()` (`pl_bolts.datamodules.kitti_datamodule.KittiDataModule` method), 232
`test_dataloader()` (`pl_bolts.datamodules.KittiDataModule` method), 209
`test_dataloader()` (`pl_bolts.datamodules.sklearn_datamodule.SklearnDataModule` method), 235
`test_dataloader()` (`pl_bolts.datamodules.SklearnDataModule` method), 211
`test_dataloader()` (`pl_bolts.datamodules.ssl_imagenet_datamodule.SSLImagenetDataModule` method), 213
`test_dataloader()` (`pl_bolts.datamodules.stl10_datamodule.STL10DataModule` method), 239
`test_dataloader()` (`pl_bolts.datamodules.STL10DataModule` method), 213

`method`), 214
`test_dataloader()` (`pl_bolts.datamodules.vision_datamodule.VisionDataModule` `method`), 241
`test_dataloader()` (`pl_bolts.models.LitMNIST` `method`), 259
`test_dataloader()` (`pl_bolts.models.mnist_module.LitMNIST` `method`), 382
`test_dataloader()` (`pl_bolts.models.rl.DQN` `method`), 290
`test_dataloader()` (`pl_bolts.models.rl.dqn_model.DQN` `method`), 312
`test_epoch_end()` (`pl_bolts.models.ImageGPT` `method`), 264
`test_epoch_end()` (`pl_bolts.models.LinearRegression` `method`), 260
`test_epoch_end()` (`pl_bolts.models.LogisticRegression` `method`), 260
`test_epoch_end()` (`pl_bolts.models.regression.linear_regression.LinearRegression` `method`), 285
`test_epoch_end()` (`pl_bolts.models.regression.LinearRegression` `method`), 283
`test_epoch_end()` (`pl_bolts.models.regression.logistic_regression.LogisticRegression` `method`), 286
`test_epoch_end()` (`pl_bolts.models.regression.LogisticRegression` `method`), 284
`test_epoch_end()` (`pl_bolts.models.rl.DQN` `method`), 290
`test_epoch_end()` (`pl_bolts.models.rl.dqn_model.DQN` `method`), 312
`test_epoch_end()` (`pl_bolts.models.vision.image_gpt.igpt_module.ImageGPT` `method`), 379
`TEST_FILE_NAME` (`pl_bolts.datasets.CIFAR10` `attribute`), 244
`TEST_FILE_NAME` (`pl_bolts.datasets.cifar10_dataset.CIFAR10` `attribute`), 250
`test_step()` (`pl_bolts.models.ImageGPT` `method`), 264
`test_step()` (`pl_bolts.models.LinearRegression` `method`), 260
`test_step()` (`pl_bolts.models.LitMNIST` `method`), 259
`test_step()` (`pl_bolts.models.LogisticRegression` `method`), 260
`test_step()` (`pl_bolts.models.mnist_module.LitMNIST` `method`), 382
`test_step()` (`pl_bolts.models.regression.linear_regression.LinearRegression` `method`), 285
`test_step()` (`pl_bolts.models.regression.LinearRegression` `method`), 284
`test_step()` (`pl_bolts.models.regression.logistic_regression.LogisticRegression` `method`), 286
`test_step()` (`pl_bolts.models.regression.LogisticRegression` `method`), 290
`test_step()` (`pl_bolts.models.vision.image_gpt.igpt_module.ImageGPT` `method`), 379
`TinyCIFAR10DataModule` (class in `pl_bolts.datamodules`), 200
`TinyCIFAR10DataModule` (class in `pl_bolts.datamodules.cifar10_datamodule`), 221
`to_device()` (`pl_bolts.callbacks.knn_online.KNNOnlineEvaluator` `method`), 193
`to_device()` (`pl_bolts.callbacks.ssl_online.SSLOnlineEvaluator` `method`), 195
`to_device()` (`pl_bolts.callbacks.SSLOnlineEvaluator` `method`), 181
`Regression` (class in `pl_bolts.models.rl.common.gym_wrappers`), 302
`train_batch()` (`pl_bolts.models.rl.DQN` `method`), 290
`train_batch()` (`pl_bolts.models.rl.noisy_dqn_model.NoisyDQN` `method`), 315
`train_batch()` (`pl_bolts.models.rl.NoisyDQN` `method`), 293
`train_batch()` (`pl_bolts.models.rl.per_dqn_model.PERDQN` `method`), 317
`train_batch()` (`pl_bolts.models.rl.PERDQN` `method`), 294
`train_batch()` (`pl_bolts.models.rl.Reinforce` `method`), 297
`train_batch()` (`pl_bolts.models.rl.reinforce_model.Reinforce` `method`), 319
`train_batch()` (`pl_bolts.models.rl.vanilla_policy_gradient_model.VanillaPolicyGradient` `method`), 322
`train_batch()` (`pl_bolts.models.rl.VanillaPolicyGradient` `method`), 299
`train_dataloader()` (`pl_bolts.datamodules.cityscapes_datamodule.CityscapesDataModule` `method`), 224
`train_dataloader()` (`pl_bolts.datamodules.CityscapesDataModule` `method`), 202
`train_dataloader()` (`pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule` `method`), 316
`train_dataloader()` (`pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule` `method`), 316

<code>(pl_bolts.datamodules.ImagenetDataModule method), 207</code>	<code>(pl_bolts.models.rl.VanillaPolicyGradient method), 299</code>
<code>train_dataloader()</code>	<code>train_dataloader()</code>
<code>(pl_bolts.datamodules.kitti_datamodule.KittiDataModule method), 232</code>	<code>(pl_bolts.models.self_supervised.AMDIM method), 324</code>
<code>train_dataloader()</code>	<code>train_dataloader()</code>
<code>(pl_bolts.datamodules.KittiDataModule method), 209</code>	<code>(pl_bolts.models.self_supervised.amdim.AMDIM method), 334</code>
<code>train_dataloader()</code>	<code>train_dataloader()</code>
<code>(pl_bolts.datamodules.sklearn_datamodule.SklearnDataModule method), 235</code>	<code>(pl_bolts.models.self_supervised.amdim.amdim_module.AMDIM method), 338</code>
<code>train_dataloader()</code>	<code>train_dataloader_labeled()</code>
<code>(pl_bolts.datamodules.SklearnDataModule method), 212</code>	<code>(pl_bolts.datamodules.stl10_datamodule.STL10DataModule method), 239</code>
<code>train_dataloader()</code>	<code>train_dataloader_labeled()</code>
<code>(pl_bolts.datamodules.ssl_imagenet_datamodule.SSLImagenetDataModule method), 237</code>	<code>(pl_bolts.datamodules.STL10DataModule method), 215</code>
<code>train_dataloader()</code>	<code>train_dataloader_mixed()</code>
<code>(pl_bolts.datamodules.SSLImagenetDataModule method), 213</code>	<code>(pl_bolts.datamodules.stl10_datamodule.STL10DataModule method), 239</code>
<code>train_dataloader()</code>	<code>train_dataloader_mixed()</code>
<code>(pl_bolts.datamodules.stl10_datamodule.STL10DataModule method), 239</code>	<code>(pl_bolts.datamodules.STL10DataModule method), 215</code>
<code>train_dataloader()</code>	<code>TRAIN_FILE_NAME</code>
<code>(pl_bolts.datamodules.STL10DataModule method), 214</code>	<code>(pl_bolts.datasets.CIFAR10 attribute), 244</code>
<code>train_dataloader()</code>	<code>TRAIN_FILE_NAME</code>
<code>(pl_bolts.datamodules.vision_datamodule.VisionDataModule method), 241</code>	<code>(pl_bolts.datasets.cifar10_dataset.CIFAR10 attribute), 250</code>
<code>train_dataloader()</code>	<code>transform()</code>
<code>(pl_bolts.datamodules.vocdetection_datamodule.VOCDetectionDataModule method), 242</code>	<code>(pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule method), 230</code>
<code>train_dataloader()</code>	<code>(pl_bolts.datamodules.ImagenetDataModule method), 207</code>
<code>(pl_bolts.datamodules.VOCDetectionDataModule method), 216</code>	<code>training_step()</code>
<code>train_dataloader()</code>	<code>(pl_bolts.models.AE method), 258</code>
<code>(pl_bolts.models.LitMNIST method), 259</code>	<code>training_step()</code>
<code>train_dataloader()</code>	<code>(pl_bolts.models.autoencoders.AE method), 265</code>
<code>(pl_bolts.models.mnist_module.LitMNIST method), 382</code>	<code>training_step()</code>
<code>train_dataloader()</code>	<code>(pl_bolts.models.autoencoders.basic_ae.basic_ae_m method), 268</code>
<code>(pl_bolts.models.rl.DQN method), 290</code>	<code>training_step()</code>
<code>train_dataloader()</code>	<code>(pl_bolts.models.autoencoders.basic_vae.basic_vae method), 270</code>
<code>(pl_bolts.models.rl.dqn_model.DQN method), 312</code>	<code>training_step()</code>
<code>train_dataloader()</code>	<code>(pl_bolts.models.autoencoders.VAE method), 266</code>
<code>(pl_bolts.models.rl.Reinforce method), 297</code>	<code>training_step()</code>
<code>train_dataloader()</code>	<code>(pl_bolts.models.detection.faster_rcnn.faster_rcnn_r method), 277</code>
<code>(pl_bolts.models.rl.reinforce_model.Reinforce method), 319</code>	<code>training_step()</code>
<code>train_dataloader()</code>	<code>(pl_bolts.models.detection.faster_rcnn.FasterRCNN method), 275</code>
<code>(pl_bolts.models.rl.vanilla_policy_gradient_model.VanillaPolicyGradient method), 322</code>	<code>training_step()</code>
<code>train_dataloader()</code>	<code>(pl_bolts.models.detection.FasterRCNN method), 272</code>
	<code>training_step()</code>
	<code>(pl_bolts.models.gans.basic.basic_gan_module.GAN method), 280</code>
	<code>training_step()</code>
	<code>(pl_bolts.models.gans.DCGAN method), 281</code>
	<code>training_step()</code>
	<code>(pl_bolts.models.gans.dcgan.dcgan_module.DCGAN method), 283</code>

`training_step()` (`pl_bolts.models.gans.GAN` `training_step()` (`pl_bolts.models.self_supervised.CPCV2`
`method`), 278 `method`), 326
`training_step()` (`pl_bolts.models.ImageGPT` `training_step()` (`pl_bolts.models.self_supervised.moco.moco2_modu`
`method`), 264 `method`), 356
`training_step()` (`pl_bolts.models.LinearRegression` `training_step()` (`pl_bolts.models.self_supervised.MocoV2`
`method`), 260 `method`), 327
`training_step()` (`pl_bolts.models.LitMNIST` `training_step()` (`pl_bolts.models.self_supervised.SimCLR`
`method`), 259 `method`), 328
`training_step()` (`pl_bolts.models.LogisticRegression` `training_step()` (`pl_bolts.models.self_supervised.simclr.simclr_modu`
`method`), 260 `method`), 359
`training_step()` (`pl_bolts.models.mnist_module.LitMNIST` `training_step()` (`pl_bolts.models.self_supervised.SimSiam`
`method`), 382 `method`), 330
`training_step()` (`pl_bolts.models.regression.linear_regression.LinearRegression` (`pl_bolts.models.self_supervised.simsiam.simsiam_m`
`method`), 285 `method`), 363
`training_step()` (`pl_bolts.models.regression.LinearRegression` `training_step()` (`pl_bolts.models.self_supervised.ssl_finetuner.SSLEFi`
`method`), 284 `method`), 375
`training_step()` (`pl_bolts.models.regression.logistic_regression.LogisticRegression` (`pl_bolts.models.self_supervised.SSLEFineTuner`
`method`), 286 `method`), 331
`training_step()` (`pl_bolts.models.regression.LogisticRegression` `training_step()` (`pl_bolts.models.self_supervised.SwAV`
`method`), 284 `method`), 333
`training_step()` (`pl_bolts.models.rl.double_dqn_model.DoubleDQN` `training_step()` (`pl_bolts.models.self_supervised.swav.SwAV`
`method`), 309 `method`), 365
`training_step()` (`pl_bolts.models.rl.DoubleDQN` `training_step()` (`pl_bolts.models.self_supervised.swav.swav_module.`
`method`), 288 `method`), 369
`training_step()` (`pl_bolts.models.rl.DQN` `method`), `training_step()` (`pl_bolts.models.SemSegment`
290 `method`), 261
`training_step()` (`pl_bolts.models.rl.dqn_model.DQN` `training_step()` (`pl_bolts.models.VAE` `method`),
`method`), 312 259
`training_step()` (`pl_bolts.models.rl.per_dqn_model.PERDQN` `training_step()` (`pl_bolts.models.vision.image_gpt.igpt_module.Imag`
`method`), 317 `method`), 380
`training_step()` (`pl_bolts.models.rl.PERDQN` `training_step()` (`pl_bolts.models.vision.segmentation.SemSegment`
`method`), 295 `method`), 381
`training_step()` (`pl_bolts.models.rl.Reinforce` `training_step()` (`pl_bolts.models.vision.SemSegment`
`method`), 297 `method`), 376
`training_step()` (`pl_bolts.models.rl.reinforce_model.Reinforce` `training_step_end()`
`method`), 319 (`pl_bolts.models.self_supervised.AMDIM`
`training_step()` (`pl_bolts.models.rl.vanilla_policy_gradient_model.VanillaPolicyGradient`
`method`), 322 `training_step_end()`
`training_step()` (`pl_bolts.models.rl.VanillaPolicyGradient` (`pl_bolts.models.self_supervised.amdim.AMDIM`
`method`), 299 `method`), 334
`training_step()` (`pl_bolts.models.self_supervised.AMDIM` `training_step_end()`
`method`), 324 (`pl_bolts.models.self_supervised.amdim.amdim_module.AMDIM`
`training_step()` (`pl_bolts.models.self_supervised.amdim.AMDIM` `method`), 338
`method`), 334 `TrainingDataMonitor` (`class` in `pl_bolts.callbacks`),
`training_step()` (`pl_bolts.models.self_supervised.amdim.amdim_module.AMDIM`
`method`), 338 `TrainingDataMonitor` (`class` in
`training_step()` (`pl_bolts.models.self_supervised.BYOL` `pl_bolts.callbacks.data_monitor`), 191
`method`), 325 `TrialCIFAR10` (`class` in `pl_bolts.datasets`), 244
`training_step()` (`pl_bolts.models.self_supervised.byol_module.BYOL` (`class` in
`method`), 344 `pl_bolts.datasets.cifar10_dataset`), 250
`training_step()` (`pl_bolts.models.self_supervised.cpc.cpc_module.CPCV2`
`method`), 350
`training_step()` (`pl_bolts.models.self_supervised.cpc.GPCV2` (`class` in `pl_bolts.models`), 262
`method`), 345 `UNet` (`class` in `pl_bolts.models.vision`), 376

UNet (class in `pl_bolts.models.vision.unet`), 381

UnlabeledImagenet (class in `pl_bolts.datasets`), 247

UnlabeledImagenet (class in `pl_bolts.datasets.imagenet_dataset`), 254

Up (class in `pl_bolts.models.vision.unet`), 382

`update_beta()` (`pl_bolts.models.rl.common.memory.PERBuffer` method), 235

`update_beta()` (`pl_bolts.models.rl.common.memory.PERBuffer` method), 304

`update_env_stats()` (`pl_bolts.datamodules.experience_source.ExperienceSource` method), 226

`update_env_stats()` (`pl_bolts.datamodules.ExperienceSource` method), 204

`update_epsilon()` (`pl_bolts.models.rl.common.agents.ValueAgent` method), 239

`update_epsilon()` (`pl_bolts.models.rl.common.agents.ValueAgent` method), 300

`update_history_queue()` (`pl_bolts.datamodules.experience_source.ExperienceSource` method), 226

`update_history_queue()` (`pl_bolts.datamodules.ExperienceSource` method), 204

`update_history_queue()` (`pl_bolts.models.rl.common.memory.MultiStepBuffer` method), 304

`update_p()` (`pl_bolts.optimizers.lars_scheduling.LARSWrapper` method), 389

`update_p()` (`pl_bolts.optimizers.LARSWrapper` method), 387

`update_priorities()` (`pl_bolts.models.rl.common.memory.PERBuffer` method), 305

`update_tau()` (`pl_bolts.callbacks.byol_updates.BYOLMAWeightUpdate` method), 189

`update_tau()` (`pl_bolts.callbacks.BYOLMAWeightUpdate` method), 178

`update_weights()` (`pl_bolts.callbacks.byol_updates.BYOLMAWeightUpdate` method), 190

`update_weights()` (`pl_bolts.callbacks.BYOLMAWeightUpdate` method), 178

V

VAE (class in `pl_bolts.models`), 258

VAE (class in `pl_bolts.models.autoencoders`), 265

VAE (class in `pl_bolts.models.autoencoders.basic_vae.basic_vae_module`), 269

`val_dataloader()` (`pl_bolts.datamodules.cityscapes_datamodule.CityscapesDataModule` method), 224

`val_dataloader()` (`pl_bolts.datamodules.CityscapesDataModule` method), 202

`val_dataloader()` (`pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule` method), 231

`val_dataloader()` (`pl_bolts.datamodules.ImagenetDataModule` method), 208

`val_dataloader()` (`pl_bolts.datamodules.kitti_datamodule.KittiDataModule` method), 232

`val_dataloader()` (`pl_bolts.datamodules.KittiDataModule` method), 209

`val_dataloader()` (`pl_bolts.datamodules.sklearn_datamodule.SklearnDataModule` method), 237

`val_dataloader()` (`pl_bolts.datamodules.SklearnDataModule` method), 212

`val_dataloader()` (`pl_bolts.datamodules.ssl_imagenet_datamodule.SSLImagenetDataModule` method), 213

`val_dataloader()` (`pl_bolts.datamodules.SSLImagenetDataModule` method), 239

`val_dataloader()` (`pl_bolts.datamodules.STL10_datamodule.STL10DataModule` method), 215

`val_dataloader()` (`pl_bolts.datamodules.vision_datamodule.VisionDataModule` method), 241

`val_dataloader()` (`pl_bolts.datamodules.vocdetection_datamodule.VOCDetectionDataModule` method), 242

`val_dataloader()` (`pl_bolts.datamodules.VOCDetectionDataModule` method), 216

`val_dataloader()` (`pl_bolts.models.LitMNIST` method), 259

`val_dataloader()` (`pl_bolts.models.mnist_module.LitMNIST` method), 382

`val_dataloader()` (`pl_bolts.models.self_supervised.AMDIM` method), 324

`val_dataloader()` (`pl_bolts.models.self_supervised.amdim.AMDIM` method), 334

`val_dataloader()` (`pl_bolts.models.self_supervised.amdim.amdim_module` method), 338

`val_dataloader_labeled()` (`pl_bolts.datamodules.stl10_datamodule.STL10DataModule` method), 239

`val_dataloader_labeled()` (`pl_bolts.datamodules.STL10DataModule` method), 215

`val_dataloader_mixed()` (`pl_bolts.datamodules.stl10_datamodule.STL10DataModule` method), 239

`val_dataloader_mixed()` (`pl_bolts.datamodules.STL10DataModule` method), 215

`val_transform()` (`pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule` method), 231

`val_transform()` (`pl_bolts.datamodules.ImagenetDataModule` method), 208

`validation_epoch_end()` (`pl_bolts.models.detection.faster_rcnn.faster_rcnn_module.FasterRCNN` method), 277

`validation_epoch_end()` (`pl_bolts.models.detection.faster_rcnn.FasterRCNN` method), 275

<code>validation_epoch_end()</code> (<i>pl_bolts.models.detection.FasterRCNN method</i>), 272	<code>validation_step()</code> (<i>pl_bolts.models.autoencoders.basic_ae.basic_ae_module.AE method</i>), 268
<code>validation_epoch_end()</code> (<i>pl_bolts.models.ImageGPT method</i>), 264	<code>validation_step()</code> (<i>pl_bolts.models.autoencoders.basic_vae.basic_vae_module.VAE method</i>), 270
<code>validation_epoch_end()</code> (<i>pl_bolts.models.LinearRegression method</i>), 260	<code>validation_step()</code> (<i>pl_bolts.models.autoencoders.VAE method</i>), 266
<code>validation_epoch_end()</code> (<i>pl_bolts.models.LogisticRegression method</i>), 260	<code>validation_step()</code> (<i>pl_bolts.models.detection.faster_rcnn.faster_rcnn_module.FasterRCNN method</i>), 277
<code>validation_epoch_end()</code> (<i>pl_bolts.models.regression.linear_regression.LinearRegression method</i>), 285	<code>validation_step()</code> (<i>pl_bolts.models.detection.faster_rcnn.FasterRCNN method</i>), 275
<code>validation_epoch_end()</code> (<i>pl_bolts.models.regression.LinearRegression method</i>), 284	<code>validation_step()</code> (<i>pl_bolts.models.detection.FasterRCNN method</i>), 272
<code>validation_epoch_end()</code> (<i>pl_bolts.models.regression.logistic_regression.LogisticRegression method</i>), 286	<code>validation_step()</code> (<i>pl_bolts.models.ImageGPT method</i>), 264
<code>validation_epoch_end()</code> (<i>pl_bolts.models.regression.LogisticRegression method</i>), 284	<code>validation_step()</code> (<i>pl_bolts.models.LinearRegression method</i>), 260
<code>validation_epoch_end()</code> (<i>pl_bolts.models.self_supervised.AMDIM method</i>), 324	<code>validation_step()</code> (<i>pl_bolts.models.LitMNIST method</i>), 259
<code>validation_epoch_end()</code> (<i>pl_bolts.models.self_supervised.amdim.AMDIM method</i>), 334	<code>validation_step()</code> (<i>pl_bolts.models.LogisticRegression method</i>), 261
<code>validation_epoch_end()</code> (<i>pl_bolts.models.self_supervised.amdim.amdim_module.AMDIM method</i>), 338	<code>validation_step()</code> (<i>pl_bolts.models.mnist_module.LitMNIST method</i>), 382
<code>validation_epoch_end()</code> (<i>pl_bolts.models.self_supervised.moco.moco2_module.MocoV2 method</i>), 356	<code>validation_step()</code> (<i>pl_bolts.models.regression.linear_regression.LinearRegression method</i>), 285
<code>validation_epoch_end()</code> (<i>pl_bolts.models.self_supervised.MocoV2 method</i>), 327	<code>validation_step()</code> (<i>pl_bolts.models.regression.LinearRegression method</i>), 284
<code>validation_epoch_end()</code> (<i>pl_bolts.models.SemSegment method</i>), 261	<code>validation_step()</code> (<i>pl_bolts.models.regression.logistic_regression.LogisticRegression method</i>), 286
<code>validation_epoch_end()</code> (<i>pl_bolts.models.vision.image_gpt.igpt_module.ImageGPT method</i>), 380	<code>validation_step()</code> (<i>pl_bolts.models.regression.LogisticRegression method</i>), 284
<code>validation_epoch_end()</code> (<i>pl_bolts.models.vision.segmentation.SemSegment method</i>), 381	<code>validation_step()</code> (<i>pl_bolts.models.self_supervised.AMDIM method</i>), 324
<code>validation_epoch_end()</code> (<i>pl_bolts.models.vision.SemSegment method</i>), 376	<code>validation_step()</code> (<i>pl_bolts.models.self_supervised.amdim.AMDIM method</i>), 334
<code>validation_step()</code> (<i>pl_bolts.models.AE method</i>), 258	<code>validation_step()</code> (<i>pl_bolts.models.self_supervised.amdim.amdim_module.AMDIM method</i>), 338
<code>validation_step()</code> (<i>pl_bolts.models.autoencoders.AE method</i>), 265	<code>validation_step()</code> (<i>pl_bolts.models.self_supervised.BYOL method</i>), 338

method), 325
 validation_step() (pl_bolts.models.self_supervised.byol.byol_module.BYOL method), 344
 validation_step() (pl_bolts.models.self_supervised.cpc.cpc_module.CPC method), 350
 validation_step() (pl_bolts.models.self_supervised.cpc.CPCV2 method), 345
 validation_step() (pl_bolts.models.self_supervised.CPCV2 method), 326
 validation_step() (pl_bolts.models.self_supervised.moco.moco2_module.MocoV2 method), 356
 validation_step() (pl_bolts.models.self_supervised.MocoV2 method), 327
 validation_step() (pl_bolts.models.self_supervised.SimCLR method), 328
 validation_step() (pl_bolts.models.self_supervised.simclr.simclr_module.SimCLR method), 359
 validation_step() (pl_bolts.models.self_supervised.SimSiam method), 330
 validation_step() (pl_bolts.models.self_supervised.simsiam.simsiam_module.SimSiam method), 363
 validation_step() (pl_bolts.models.self_supervised.ssl_finetuner.SSLFineTuner method), 375
 validation_step() (pl_bolts.models.self_supervised.SSLFineTuner method), 331
 validation_step() (pl_bolts.models.self_supervised.SwAV method), 333
 validation_step() (pl_bolts.models.self_supervised.swav.SwAV method), 365
 validation_step() (pl_bolts.models.self_supervised.swav.swav_module.SwAV method), 369
 validation_step() (pl_bolts.models.SemSegment method), 261
 validation_step() (pl_bolts.models.VAE method), 259
 validation_step() (pl_bolts.models.vision.image_gpt.igpt_module.ImageGPT method), 380
 validation_step() (pl_bolts.models.vision.segmentation.SemSegment method), 381
 validation_step() (pl_bolts.models.vision.SemSegment method), 376
 VanillaPolicyGradient (class in pl_bolts.models.rl.common.agents), 300
 VanillaPolicyGradient (class in pl_bolts.models.rl), 297
 VanillaPolicyGradient (class in pl_bolts.models.rl.vanilla_policy_gradient_model), 320
 VerificationBase (class in pl_bolts.callbacks.verifcation.base), 183
 VisionCallbackBase (class in pl_bolts.callbacks.verifcation.base), 184
 VisionDataModule (class in pl_bolts.datamodules.vision_datamodule), 240
 VOCDetectionDataModule (class in pl_bolts.datamodules), 215
 VOCDetectionDataModule (class in pl_bolts.datamodules.vocdetection_datamodule), 215

W

warning_message() (pl_bolts.callbacks.verifcation.base.VerificationCallbackBase method), 185
 warning_message() (in module pl_bolts.models.self_supervised.resnets), 373
 warning_message() (in module pl_bolts.models.self_supervised.resnets), 373