PyTorch-Lightning-Bolts Documentation

Release 0.1.1

PyTorchLightning et al.

Sep 09, 2020

START HERE

CHAPTER

ONE

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(...)
)
```

1.1 Community Built

Bolts are built-by the Lightning community and contributed 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.

1.1.1 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!

1.1.2 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

---

1.2 When to use Bolts

1.2.1 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(pretrained='imagenet2012')
encoder = model1.encoder
encoder.freeze()

# bolts are pretrained on different datasets
model2 = CPCV2(encoder='resnet18', pretrained='imagenet128').freeze()
model3 = CPCV2(encoder='resnet18', pretrained='stl10').freeze()
```

(continues on next page)

(continued from previous page)

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

# which is better?
```

1.2.2 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.freeze()

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()
```

1.2.3 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):

    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()
```

(continues on next page)

(continued from previous page)

```
        return loss
```

1.3 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()])
```

1.4 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)
```

1.5 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)
)
```

1.5.1 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
```

(continues on next page)

(continued from previous page)

```
model = LinearRegression()
trainer = pl.Trainer(num_gpus=8)
trainer.fit(model, loaders.train_dataloader(), 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)
```

1.5.2 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)

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

# fit
trainer = pl.Trainer(tpu_cores=8, precision=16)
trainer.fit(model, dm.train_dataloader(), dm.val_dataloader())

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

Any input will be flattened across all dimensions except the firs 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
```

(continues on next page)

(continued from previous page)

```
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)
```

1.6 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 :)

1.7 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
```

CHAPTER

TWO

MODEL QUALITY CONTROL

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

2.1 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!

2.2 Contribution requirements

2.2.1 Benchmarked

Models have known performance results on common baseline datasets.

2.2.2 Device agnostic

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

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

2.2.3 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)
```

2.2.4 Tested

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

- Live build
- Tests

2.2.5 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 = ...
```

2.2.6 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.
- Hyperparameters configurations for the results.

Thank you for all your amazing contributions!

2.3 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!

2.4 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.

CHAPTER

THREE

BUILD A CALLBACK

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

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

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

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

3.1 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.

3.2 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.

CHAPTER

FOUR

INFO CALLBACKS

These callbacks give all sorts of useful information during training.

4.1 Print Table Metrics

This callbacks 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
```

CHAPTER

FIVE

SELF-SUPERVISED CALLBACKS

Useful callbacks for self-supervised learning models

5.1 BYOLMAWeightUpdate

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

class pl_bolts.callbacks.self_supervised.**BYOLMAWeightUpdate**(*initial_tau=0.996*)

Bases: pytorch_lightning.Callback

Weight update rule from BYOL.

Your model should have a:

- 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:

```
from pl_bolts.callbacks.self_supervised import BYOLMAWeightUpdate

# model must have 2 attributes
model = Model()
model.online_network = ...
model.target_network = ...

# make sure to set max_steps in Trainer
trainer = Trainer(callbacks=[BYOLMAWeightUpdate()], max_steps=1000)
```

Parameters **initial_tau** – starting tau. Auto-updates with every training step

5.2 SSLOnlineEvaluator

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

class pl_bolts.callbacks.self_supervised.**SSLOnlineEvaluator**(*drop_p=0.2*, *hidden_dim=1024*, *z_dim=None*, *num_classes=None*)

Bases: pytorch_lightning.Callback

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

Example:

```
from pl_bolts.callbacks.self_supervised import SSLOnlineEvaluator

# your model must have 2 attributes
model = Model()
model.z_dim = ... # the representation dim
model.num_classes = ... # the num of classes in the model
```

Parameters

- **drop_p** (float) – (0.2) dropout probability
- **hidden_dim** (int) –

(1024) the hidden dimension for the finetune MLP

get_representations(*pl_module*, *x*)

Override this to customize for the particular model :param _sphinx_paramlinks_pl_bolts.callbacks.self_supervised.SSLOnlin
:param _sphinx_paramlinks_pl_bolts.callbacks.self_supervised.SSLOnlineEvaluator.get_representations.x:

CHAPTER

SIX

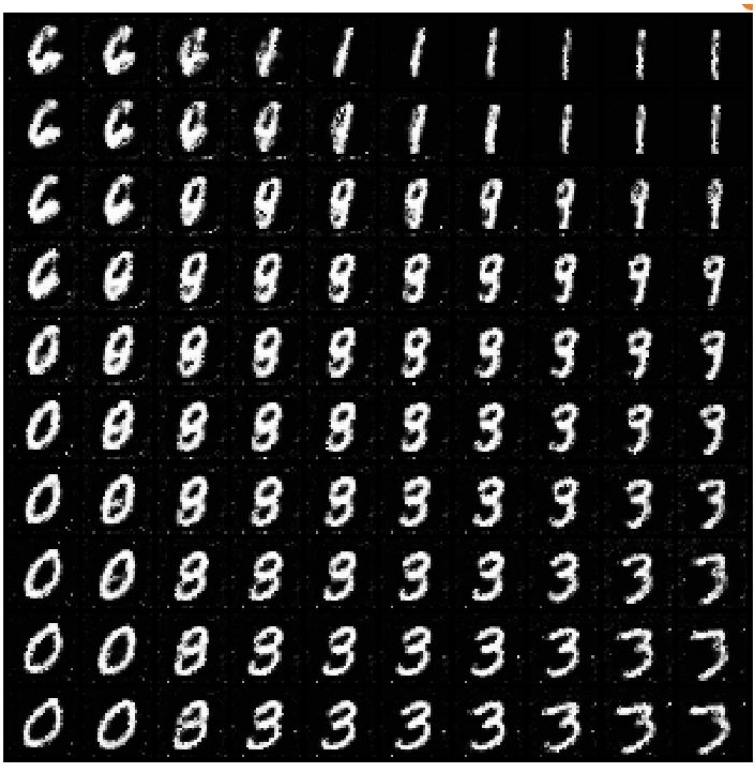
VARIATIONAL CALLBACKS

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

6.1 Latent Dim Interpolator

Interpolates latent dims.

Example output:



class pl_bolts.callbacks.variational.**LatentDimInterpolator**(*interpolate_epoch_interval=20*, *range_start=-5*, *range_end=5*, *num_samples=2*)

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** –
- **range_start** – default -5
- **range_end** – default 5
- **num_samples** – default 2

CHAPTER

SEVEN

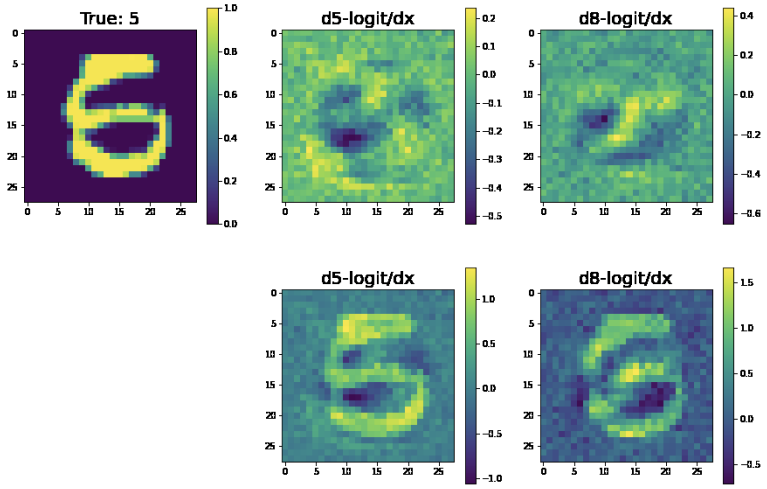
VISION CALLBACKS

Useful callbacks for vision models

7.1 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*, *projection_factor=3*, *min_logit_value=5.0*, *logging_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

Parameters

- **top_k** – How many "offending" images we should plot
- **projection_factor** – How much to multiply the input image to make it look more like this logit label
- **min_logit_value** – Only consider logit values above this threshold
- **logging_batch_interval** – how frequently to inspect/potentially plot something
- **max_logit_difference** – when the top 2 logits are within this threshold we consider them confused

Authored by:

- Alfredo Canziani

7.2 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()])
```

CHAPTER

EIGHT

DATAMODULES

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

- Downloading/preparing data.
- Train/val/test splits.
- Transforms

Then you can use it like this:

Example:

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

trainer = Trainer()
trainer.fit(model, 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

CHAPTER

NINE

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)
```

9.1 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.

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

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
```

9.2 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*, *random_state=1234*, *shuffle=True*, **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)
...
>>> # train set
>>> train_loader = loaders.train_dataloader(batch_size=32)
>>> len(train_loader.dataset)
355
>>> len(train_loader)
11
>>> # validation set
>>> val_loader = loaders.val_dataloader(batch_size=32)
>>> len(val_loader.dataset)
100
```

(continues on next page)

(continued from previous page)

```
>>> len(val_loader)
3
>>> # test set
>>> test_loader = loaders.test_dataloader(batch_size=32)
>>> len(test_loader.dataset)
51
>>> len(test_loader)
1
```

CHAPTER

TEN

VISION DATAMODULES

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

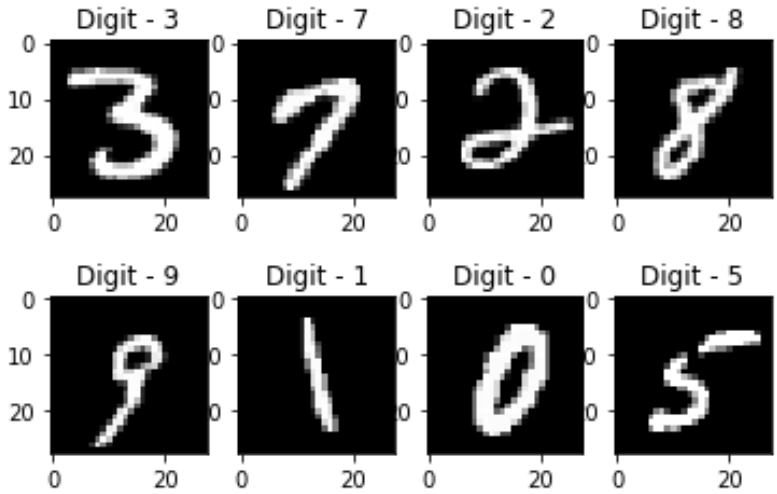
10.1 Supervised learning

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

10.1.1 BinaryMNIST

class pl_bolts.datamodules.binary_mnist_datamodule.**BinaryMNISTDataModule**(*data_dir*, *val_split=5000*, *num_workers=16*, *normalize=False*, *seed=42*, **args*, ***kwargs*)

Bases: pytorch_lightning.LightningDataModule



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, dm)
```

Parameters

- **data_dir** (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
- **normalize** (bool) – If true applies image normalize

prepare_data()
Saves MNIST files to data_dir

test_dataloader(*batch_size=32*, *transforms=None*)
MNIST test set uses the test split

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

train_dataloader(*batch_size=32*, *transforms=None*)
MNIST train set removes a subset to use for validation

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

val_dataloader(*batch_size=32*, *transforms=None*)
MNIST val set uses a subset of the training set for validation

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

property num_classes
Return: 10

10.1.2 CityScapes

class pl_bolts.datamodules.cityscapes_datamodule.**CityscapesDataModule**(*data_dir*, *val_split=5000*, *num_workers=16*, *batch_size=32*, *seed=42*, **args*, ***kwargs*)

Bases: pytorch_lightning.LightningDataModule

Standard Cityscapes, train, val, test splits and transforms

Specs:

- 30 classes (road, person, sidewalk, etc...)
- (image, target) - image dims: (3 x 32 x 32), target dims: (3 x 32 x 32)

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, dm)
```

Or you can set your own transforms

Example:

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

Parameters

- **data_dir** – where to save/load the data
- **val_split** – how many of the training images to use for the validation split
- **num_workers** – how many workers to use for loading data
- **batch_size** – number of examples per training/eval step

prepare_data()
Saves Cityscapes files to data_dir

test_dataloader()
Cityscapes test set uses the test split

train_dataloader()
Cityscapes train set with removed subset to use for validation

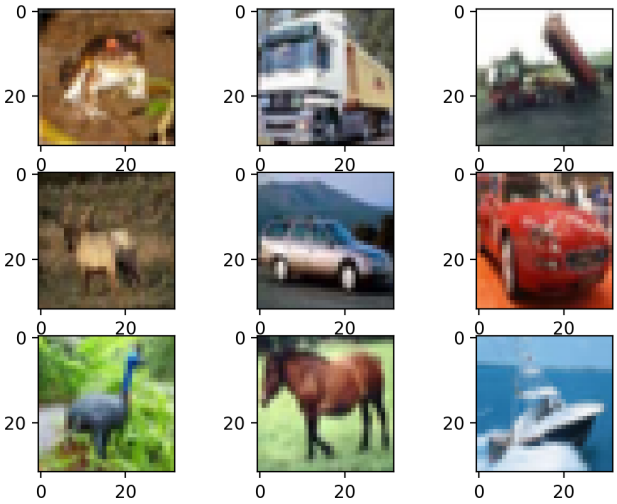
val_dataloader()
Cityscapes val set uses a subset of the training set for validation

property num_classes
Return: 30

10.1.3 CIFAR-10

class pl_bolts.datamodules.cifar10_datamodule.**CIFAR10DataModule**(*data_dir=None*, *val_split=5000*, *num_workers=16*, *batch_size=32*, *seed=42*, **args*, ***kwargs*)

Bases: pytorch_lightning.LightningDataModule



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, 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** (int) – how many of the training images to use for the validation split
- **num_workers** (int) – how many workers to use for loading data
- **batch_size** (int) – number of examples per training/eval step

prepare_data()
Saves CIFAR10 files to data_dir

test_dataloader()
CIFAR10 test set uses the test split

train_dataloader()
CIFAR train set removes a subset to use for validation

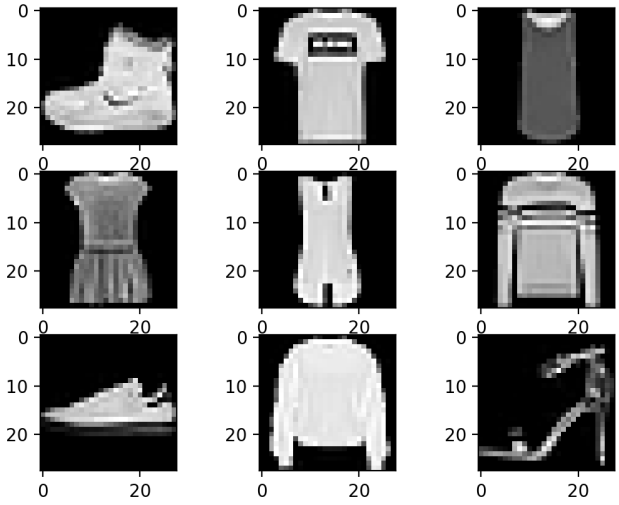
val_dataloader()
CIFAR10 val set uses a subset of the training set for validation

property num_classes
Return: 10

10.1.4 FashionMNIST

class pl_bolts.datamodules.fashion_mnist_datamodule.**FashionMNISTDataModule**(*data_dir*, *val_split=5000*, *num_workers=16*, *seed=42*, **args*, ***kwargs*)

Bases: pytorch_lightning.LightningDataModule



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, dm)
```

Parameters

- **data_dir** (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

prepare_data()
Saves FashionMNIST files to data_dir

test_dataloader(*batch_size=32*, *transforms=None*)
FashionMNIST test set uses the test split

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

train_dataloader(*batch_size=32*, *transforms=None*)
FashionMNIST train set removes a subset to use for validation

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

val_dataloader(*batch_size=32*, *transforms=None*)
FashionMNIST val set uses a subset of the training set for validation

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

property num_classes
Return: 10

10.1.5 Imagenet

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*, **args*, ***kwargs*)

Bases: pytorch_lightning.LightningDataModule

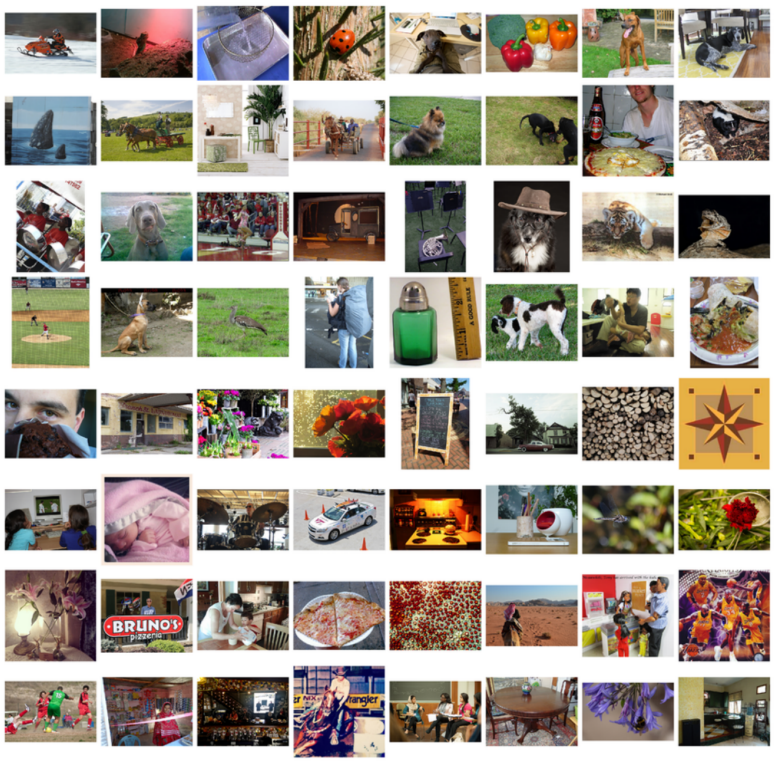
Specs:

- 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(IMAGENET_PATH)
model = LitModel()

Trainer().fit(model, 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

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.

test_dataloader()

Uses the validation split of imagenet2012 for testing

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

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]
    ),
])
```

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

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]
    ),
])
```

property num_classes
Return:

1000

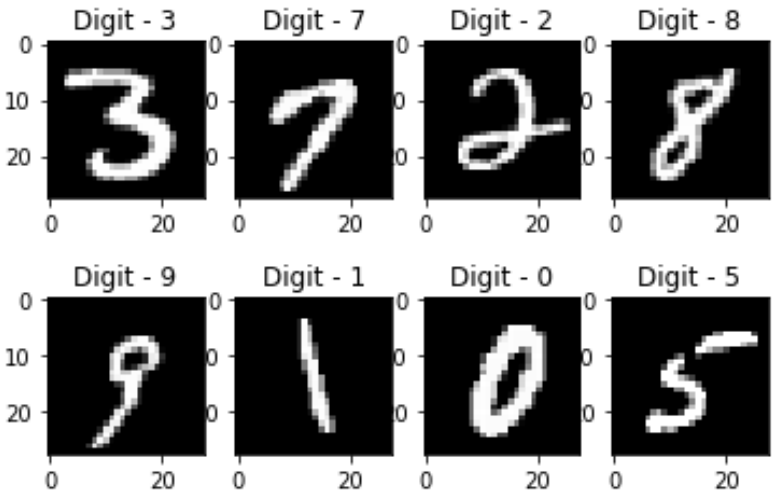
10.1.6 MNIST

class pl_bolts.datamodules.mnist_datamodule.**MNISTDataModule**(*data_dir*, *val_split=5000*, *num_workers=16*, *normalize=False*, *seed=42*, **args*, ***kwargs*)

Bases: pytorch_lightning.LightningDataModule

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, dm)
```

Parameters

- **data_dir** (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
- **normalize** (bool) – If true applies image normalize

prepare_data()
Saves MNIST files to data_dir

test_dataloader(*batch_size=32*, *transforms=None*)
MNIST test set uses the test split

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

train_dataloader(*batch_size=32*, *transforms=None*)
MNIST train set removes a subset to use for validation

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

val_dataloader(*batch_size=32*, *transforms=None*)
MNIST val set uses a subset of the training set for validation

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

property num_classes
Return: 10

10.2 Semi-supervised learning

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

10.2.1 Imagenet (ssl)

class pl_bolts.datamodules.ssl_imagenet_datamodule.**SSLImagenetDataModule**(*data_dir*, *meta_dir=None*, *num_workers=16*, **args*, ***kwargs*)

Bases: pytorch_lightning.LightningDataModule

10.2.2 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*, **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, 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

prepare_data()
Downloads the unlabeled, train and test split

test_dataloader()
Loads the test split of STL10

Parameters

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

train_dataloader()
Loads the 'unlabeled' split minus a portion set aside for validation via *unlabeled_val_split*.

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

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

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

CHAPTER

ELEVEN

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** – The PyTorch Dataset or DataLoader we're using to load.
- **device** – The PyTorch device we are loading to
- **q_size** – Size of the queue used to store the data loaded to the device
- **num_batches** – 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

CHAPTER

TWELVE

DUMMYDATASET

class pl_bolts.datamodules.dummy_dataset.**DummyDataset**(**shapes*, *num_samples=10000*)

Bases: torch.utils.data.Dataset

Generate a dummy dataset

Parameters

- ***shapes** – list of shapes
- **num_samples** – how many samples to use in this dataset

Example:

```
from pl_bolts.datamodules import DummyDataset

# mnist dims
>>> ds = DummyDataset((1, 28, 28), (1,))
>>> dl = DataLoader(ds, batch_size=7)
...
>>> batch = next(iter(dl))
>>> x, y = batch
>>> x.size()
torch.Size([7, 1, 28, 28])
>>> y.size()
torch.Size([7, 1])
```

CHAPTER

THIRTEEN

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

13.1 Your Loss

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

CHAPTER

FOURTEEN

REINFORCEMENT LEARNING

These are common losses used in RL.

14.1 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

14.2 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

14.3 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

CHAPTER

FIFTEEN

BOLTS LOGGERS

The loggers in this package are being considered to be added to the main PyTorch Lightning repository. These loggers may be more unstable, in development, or not fully tested yet.

Note: This module is a work in progress

15.1 allegro.ai TRAINS

allegro.ai is a third-party logger. To use *TrainsLogger* as your logger do the following. First, install the package:

```
pip install trains
```

Then configure the logger and pass it to the `Trainer`:

```
from pl_bolts.loggers import TrainsLogger
trains_logger = TrainsLogger(
    project_name='examples',
    task_name='pytorch lightning test',
)
trainer = Trainer(logger=trains_logger)
```

```
class MyModule(LightningModule):
    def __init__(self):
        some_img = fake_image()
        self.logger.experiment.log_image('debug', 'generated_image_0', some_img, 0)
```

See also:

TrainsLogger docs.

15.1.1 Your Logger

Add your loggers here!

CHAPTER

SIXTEEN

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

16.1 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.autoencoders import VAE

model = VAE(pretrained='imagenet2012')
encoder = model.encoder
encoder.freeze()

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 resnet18 was trained using self-supervised learning (no labels) on Imagenet, and thus might perform better than the same resnet18 trained with labels

```
# trained without labels
from pl_bolts.models.self_supervised import CPCV2

model = CPCV2(encoder='resnet18', pretrained='imagenet128')
resnet18_unsupervised = model.encoder.freeze()

# trained with labels
from torchvision.models import resnet18
resnet18_supervised = resnet18(pretrained=True)

# perhaps the features when trained without labels are much better for classification␣
↪or other tasks
x = image_sample()
unsup_feats = resnet18_unsupervised(x)
sup_feats = resnet18_supervised(x)

# which one will be better?
```

Bolts are often trained on more than just one dataset.

```
model = CPCV2(encoder='resnet18', pretrained='stl10')
```

16.2 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.

16.2.1 Unfrozen Finetuning

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

```
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)
    ...
```

Or as a LightningModule

```
class FineTuner(pl.LightningModule):

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

    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(resnet18)
trainer.fit(model)
```

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

16.2.2 Freeze then unfreeze

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

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

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()
```

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(resnet18)
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):
```

(continues on next page)

(continued from previous page)

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

16.2.3 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.

```
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(hidden_dim=hd, learning_rate=lr)
        trainer = Trainer()
        trainer.fit(vae)
```

16.3 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!
trainer = Trainer(gpus=2)
trainer.fit(model, train_data, val_data)
```

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

16.4 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.

16.4.1 Extending work

Perhaps a research project requires modifying a part of a know approach. In this case, you're better off only changing that part of a system that is already know 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,
```

(continues on next page)

(continued from previous page)

```
            '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
```

16.4.2 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
```

16.4.3 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):
        suoer().__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)

        return unsup_loss + vae_loss + gan_loss
```

CHAPTER

SEVENTEEN

AUTOENCODERS

This section houses autoencoders and variational autoencoders.

17.1 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
```

class pl_bolts.models.autoencoders.**AE**(*datamodule=None*, *input_channels=1*, *input_height=28*, *input_width=28*, *latent_dim=32*, *batch_size=32*, *hidden_dim=128*, *learning_rate=0.001*, *num_workers=8*, *data_dir='.'*, ***kwargs*)

Bases: pytorch_lightning.LightningModule

Arg:

datamodule: the datamodule (train, val, test splits) input_channels: num of image channels input_height: image height input_width: image width latent_dim: emb dim for encoder batch_size: the batch size hidden_dim: the encoder dim learning_rate: the learning rate num_workers: num dataloader workers data_dir: where to store data

---

17.1.1 Variational Autoencoders

17.2 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
```

class pl_bolts.models.autoencoders.**VAE**(*hidden_dim=128*, *latent_dim=32*, *input_channels=3*, *input_width=224*, *input_height=224*, *batch_size=32*, *learning_rate=0.001*, *data_dir='.'*, *datamodule=None*, *num_workers=8*, *pretrained=None*, ***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 imagenet
vae = VAE(pretrained='imagenet')

# pretrained on cifar10
vae = VAE(pretrained='cifar10')
```

Parameters

- **hidden_dim** (int) – encoder and decoder hidden dims
- **latent_dim** (int) – latenet code dim
- **input_channels** (int) – num of channels of the input image.
- **input_width** (int) – image input width
- **input_height** (int) – image input height
- **batch_size** (int) – the batch size

- **the learning rate** (*learning_rate"*) –
- **data_dir** (str) – the directory to store data
- **datamodule** (Optional[LightningDataModule]) – The Lightning DataModule
- **pretrained** (Optional[str]) – Load weights pretrained on a dataset

CHAPTER

EIGHTEEN

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 $beta$ that minimizes the mean squared error between the predicted and true target values.

```
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, loaders.train_dataloader(), loaders.val_dataloader())
trainer.test(test_dataloaders=loaders.test_dataloader())
```

class pl_bolts.models.regression.linear_regression.**LinearRegression**(*input_dim*, *bias=True*, *learning_rate=0.0001*, *optimizer=torch.optim.Adam*, *l1_strength=None*, *l2_strength=None*, ***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+)

- **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** (Optional[float]) – L1 regularization strength (default=None)
- **l2_strength** (Optional[float]) – L2 regularization strength (default=None)

18.2 Logistic Regression

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

To leverage autograd we think of logistic regression as a one-layer neural network with a sigmoid activation. This allows us to support training on GPUs and TPUs.

```
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, dm.train_dataloader(), dm.val_dataloader())

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

Any input will be flattened across all dimensions except the firs 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}
```

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)

CHAPTER

NINETEEN

CONVOLUTIONAL ARCHITECTURES

This package lists contributed convolutional architectures.

19.1 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

19.2 Image GPT

class pl_bolts.models.vision.image_gpt.igpt_module.**ImageGPT**(*datamodule=None*, *embed_dim=16*, *heads=2*, *layers=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

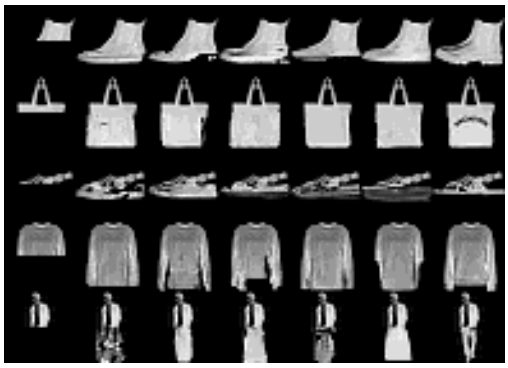
Implementation contributed by:

- 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.)
–embed_dim	16	512
–heads	2	8
–layers	8	24
–pixels	28	32
–vocab_size	16	512
–num_classes	10	10
–batch_size	64	128
–learning_rate	0.01	0.01
–steps	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

- **datamodule** (Optional[LightningDataModule]) – LightningDataModule
- **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

19.3 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])
```

CHAPTER

TWENTY

GANS

Collection of Generative Adversarial Networks

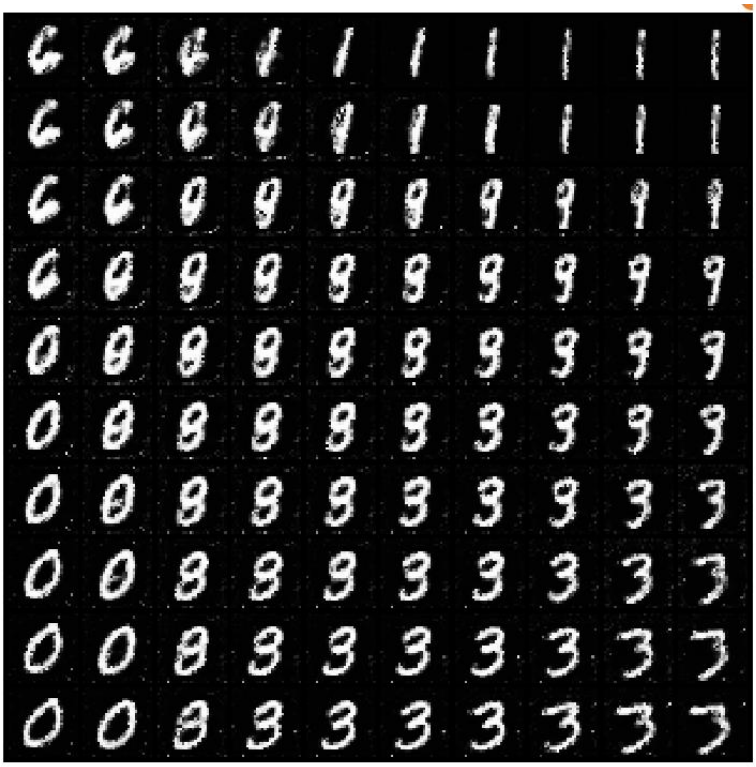
20.1 Basic GAN

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.

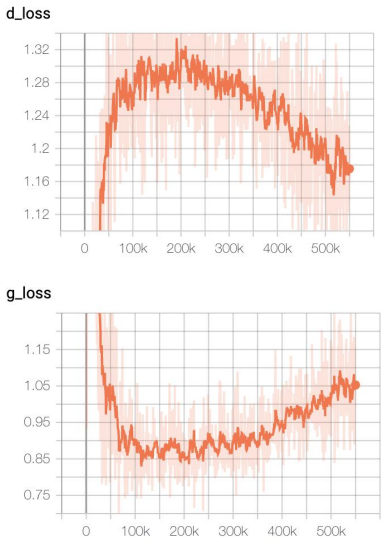
Implemented by:

- 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**(*datamodule=None*, *latent_dim=32*, *batch_size=100*, *learning_rate=0.0002*, *data_dir=''*, *num_workers=8*, ***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

- **datamodule** (Optional[LightningDataModule]) – the datamodule (train, val, test splits)
- **latent_dim** (int) – emb dim for encoder
- **batch_size** (int) – the batch size
- **learning_rate** (float) – the learning rate
- **data_dir** (str) – where to store data

- **num_workers** (int) – data workers

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)
```

CHAPTER

TWENTYONE

REINFORCEMENT LEARNING

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

21.1 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.

21.2 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 models 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.

21.2.1 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 apporximating 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 apporximate 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

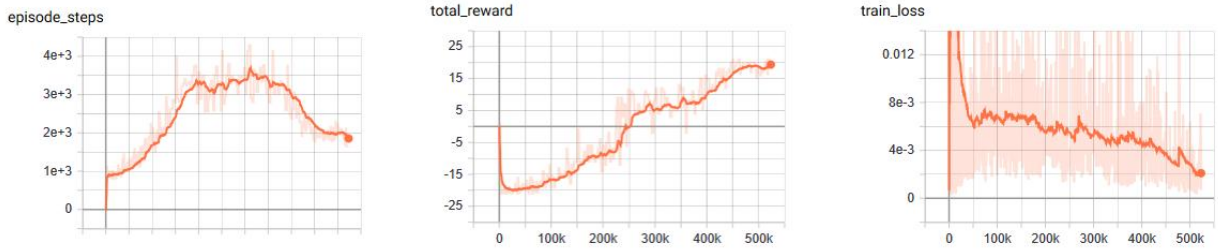
$$Q(s,a) < -(r + \gamma \max_{a' \in A} Q(s',a'))^2$$

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*, *n_steps=1*, *seed=123*, *num_envs=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)
```

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 espilon = 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
- **num_envs** (int) – number of environments to run the agent in at once

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*

_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 :type _sphinx_paramlinks_pl_bolts.models.rl.dqn_model.DQN.add_model_specific_args.arg_parser: ArgumentParser :param _sphinx_paramlinks_pl_bolts.models.rl.dqn_model.DQN.add_model_specific_args.arg_parser 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 :type _sphinx_paramlinks_pl_bolts.models.rl.dqn_model.DQN.forward.x: Tensor :param _sphinx_paramlinks_pl_bolts.models.rl.dqn_model.DQN.forward.x: environment state

Return type Tensor

Returns q values

static make_environment(*env_name*, *seed*)
Initialise gym environment

Parameters

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

Return type Env

Returns gym environment

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

Return type None

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 :rtype: 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 :type _sphinx_paramlinks_pl_bolts.models.rl.dqn_model.DQN.training_step.batch: Tuple[Tensor, Tensor] :param _sphinx_paramlinks_pl_bolts.models.rl.dqn_model.DQN.training_step.batch: current mini batch of replay data :param _sphinx_paramlinks_pl_bolts.models.rl.dqn_model.DQN.training_step._: batch number, not used

Return type OrderedDict

Returns Training loss and log metrics

21.2.2 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 "optimisitc" because it is chosen by the target network.

DQN expected return

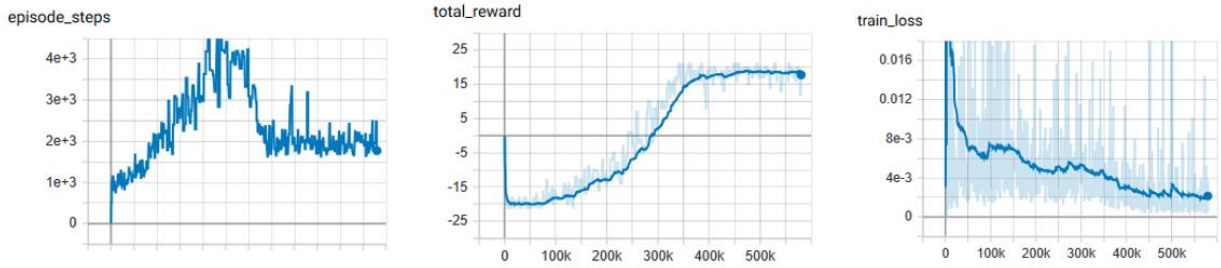
$$Q(s_t, a_t) = r_t + \gamma * \max_{Q'}(S_{t+1}, a)$$

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

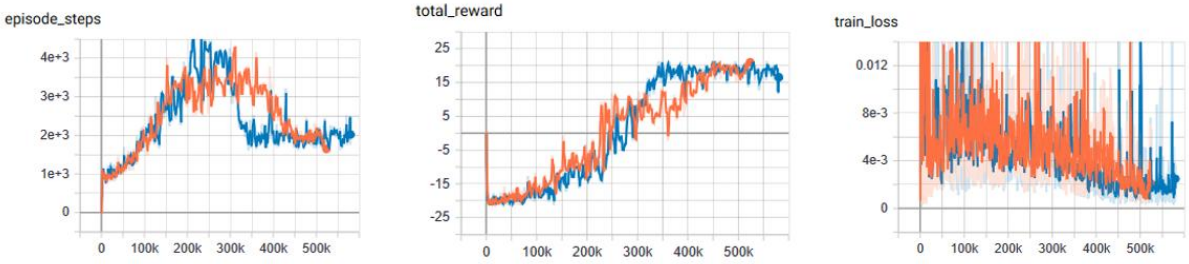
Double DQN: Pong



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*, *n_steps=1*, *seed=123*, *num_envs=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)
```

Parameters

- **env** (str) – gym environment tag
- **gpus** – number of gpus being used
- **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 espilon = eps_end
- **sync_rate** (int) – the number of iterations between syncing up the target network with the train network
- **gamma** (float) – discount factor
- **lr** – 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
- **sample_len** – the number of samples to pull from the dataset iterator and feed to the DataLoader

---

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*

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)
```

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 espilon = 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
- **num_envs** (int) – number of environments to run the agent in at once

---

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*

---

training_step(*batch*, _)

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch recieved :type _sphinx_paramlinks_pl_bolts.models.rl.double_dqn_model.DoubleDQN.training_step.batch: Tuple[Tensor, Tensor] :param _sphinx_paramlinks_pl_bolts.models.rl.double_dqn_model.DoubleDQN.training_step.l current mini batch of replay data :param _sphinx_paramlinks_pl_bolts.models.rl.double_dqn_model.DoubleDQN.training_st batch number, not used

Return type OrderedDict

Returns Training loss and log metrics

21.2.3 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 seperate estimators for state value and advantage makes the Dueling DQN robust to this type of scenario

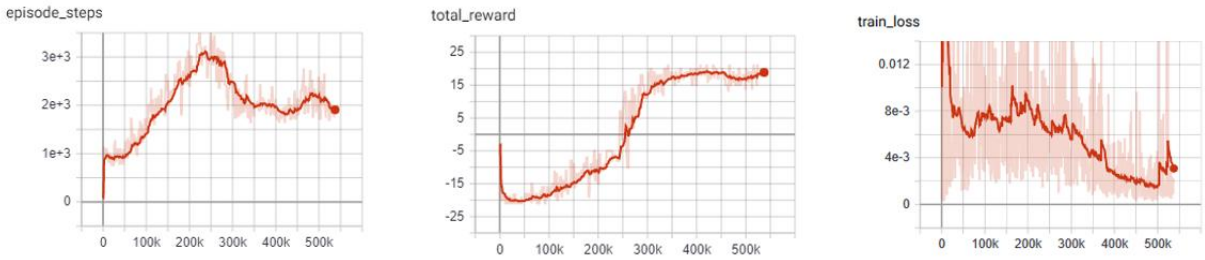
Dueling DQN Results

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 plateus.

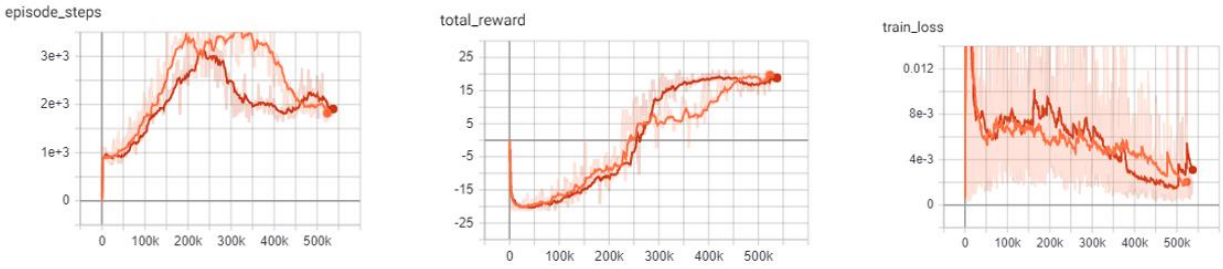


DQN vs Dueling DQN: Pong

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*, *learning_rate=0.0001*, *batch_size=32*, *replay_size=100000*, *warm_start_size=10000*, *avg_reward_len=100*, *min_episode_reward=-21*, *n_steps=1*, *seed=123*, *num_envs=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)
```

Parameters

- **env** (str) – gym environment tag
- **gpus** – number of gpus being used
- **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 espilon = eps_end
- **sync_rate** (int) – the number of iterations between syncing up the target network with the train network
- **gamma** (float) – discount factor
- **lr** – 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
- **sample_len** – the number of samples to pull from the dataset iterator and feed to the DataLoader

---

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

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)
```

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 espilon = 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
- **num_envs** (int) – number of environments to run the agent in at once

---

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*

---

build_networks()

Initializes the Dueling DQN train and target networks

Return type None

21.2.4 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 seperate 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 weightsof 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 Independant Gaussian as it has been reported there isn't much difference in results for these benchmark environments.

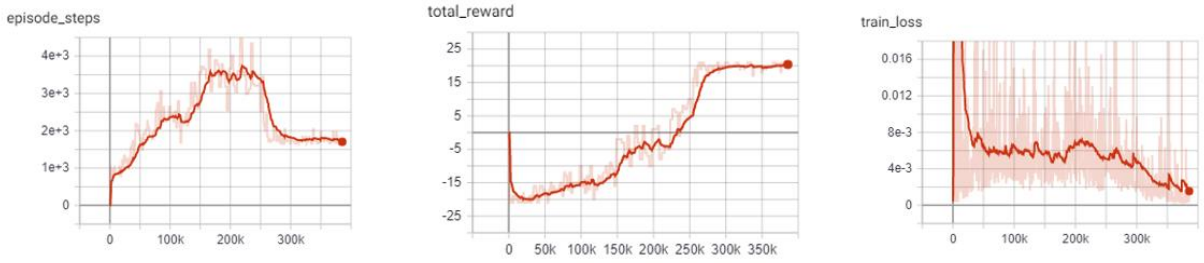
In order to update the basic DQN to a Noisy DQN we need to do the following

Noisy DQN Results

The results below improved stability and faster performance growth.

Noisy DQN baseline: Pong

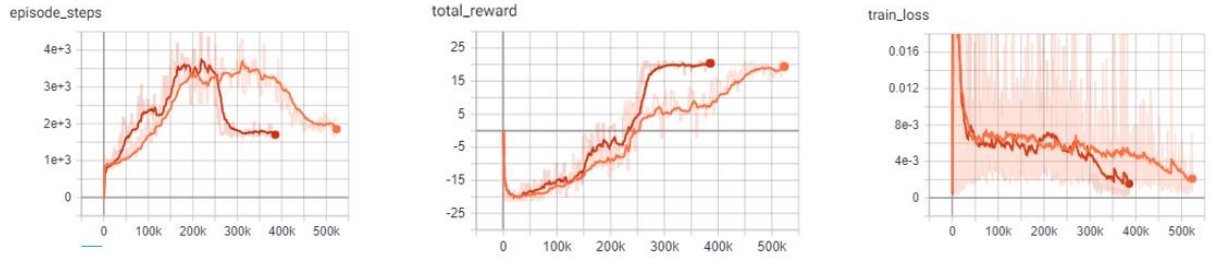
Similar to the other improvements, the average score of the agent reaches positive numbers around the 250k mark and steadily increases till convergence.



DQN vs Dueling 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*, *replay_size=100000*, *warm_start_size=10000*, *avg_reward_len=100*, *min_episode_reward=-21*, *n_steps=1*, *seed=123*, *num_envs=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)
```

Parameters

- **env** (str) – gym environment tag
- **gpus** – number of gpus being used
- **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 espilon = eps_end
- **sync_rate** (int) – the number of iterations between syncing up the target network with the train network
- **gamma** (float) – discount factor
- **lr** – 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
- **to fill the buffer with a starting point** (*training*) –
- **sample_len** – the number of samples to pull from the dataset iterator and feed to the DataLoader

---

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

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)
```

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 espilon = 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
- **num_envs** (int) – number of environments to run the agent in at once

---

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*

---

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

21.2.5 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 aproximating 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 are updates become unstable. In general, small values of 2-4 work best.

N-Step Benefits

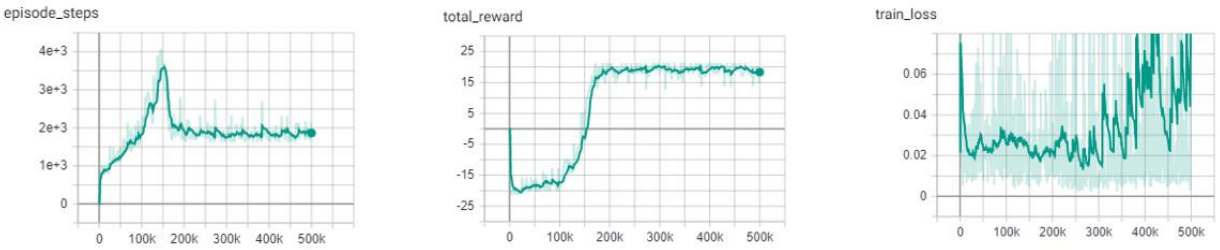
- Multi-Step learning is capable of learning faster than typical 1 step learning methods.
- **Note that this method introduces a new hyperparameter n. Although n=4 is generally a good starting point and provides** good results across the board.

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.

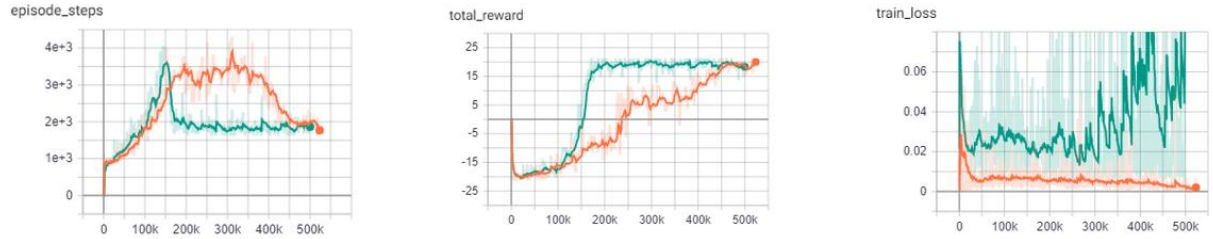
N-Step DQN: Pong

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 large 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)
```

21.2.6 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 sunrise the agent.

The priority of each sample is defined below where

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

where pi is the priority of the ith sample in the buffer and is the number that shows how much emphasis we give to the priority. If = 0 , our sampling will become uniform as in the classic DQN method. Larger values for 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

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

Where beta is a hyper parameter between 0-1. When beta is 1 the bias is fully compensated. However authors noted that in practice it is better to start beta 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(logN) for updating priorities.

Update loss function

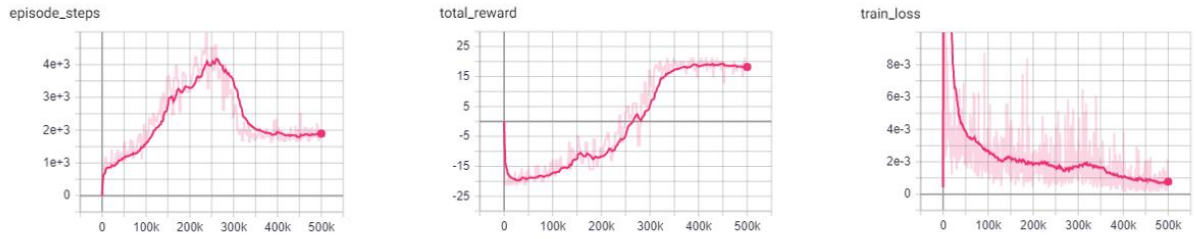
The next thing we do is to use the sample weights that we get from PER. Add the following code to the end of the loss function. This applies the weights of our sample to the batch loss. Then we return the mean loss and weighted loss for each datum, with the addition of a small epsilon value.

PER Results

The results below show improved stability and faster performance growth.

PER DQN: Pong

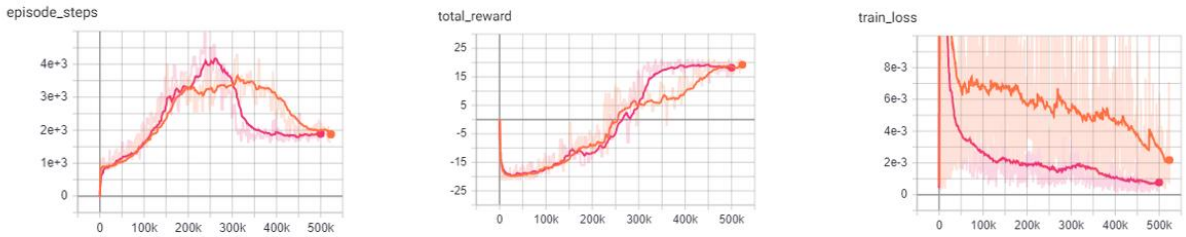
Similar to the other improvements, we see that PER improves the stability of the agents training and shows to converged on an optimal policy faster.



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 siginificantly 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*, *n_steps=1*, *seed=123*, *num_envs=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)

Args:
    env: gym environment tag
    gpus: number of gpus being used
    eps_start: starting value of epsilon for the epsilon-greedy exploration
    eps_end: final value of epsilon for the epsilon-greedy exploration
    eps_last_frame: the final frame in for the decrease of epsilon. At this frame␣
↪espilon = eps_end
    sync_rate: the number of iterations between syncing up the target network␣
↪with the train network
    gamma: discount factor
    learning_rate: learning rate
    batch_size: size of minibatch pulled from the DataLoader
    replay_size: total capacity of the replay buffer
    warm_start_size: how many random steps through the environment to be carried␣
↪out at the start of
        training to fill the buffer with a starting point
    num_samples: the number of samples to pull from the dataset iterator and feed␣
↪to the DataLoader

.. 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`
```

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)
```

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 espilon = 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
- **num_envs** (int) – number of environments to run the agent in at once

---

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*

_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 :rtype:

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

21.3 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 specifc 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 continuos 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 faste, however they are not as sample efficient and generally require more interactions with the environment.

21.3.1 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*, *num_envs=1*, *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("PongNoFrameskip-v4")
```

Train:

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

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

---

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*

_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 cargs 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]

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

21.3.2 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*, *num_envs=4*, *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("PongNoFrameskip-v4")
```

Train:

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

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

---

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*

_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 cargs added

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

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

Return type OrderedDict

Returns Training loss and log metrics

CHAPTER

TWENTYTWO

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

22.1 Use cases

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

22.1.1 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.

Example:

```
from pl_bolts.models.self_supervised import CPCV2

# load resnet18 pretrained using CPC on imagenet
model = CPCV2(pretrained='resnet18')
cpc_resnet18 = model.encoder
cpc_resnet18.freeze()

# it supports any torchvision resnet
model = CPCV2(pretrained='resnet50')
```

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 = cpc_resnet18(x)
```

22.1.2 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),
)
```

22.1.3 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)
```

22.2 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.

22.2.1 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*, ***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

22.2.2 BYOL

class pl_bolts.models.self_supervised.**BYOL**(*num_classes*, *learning_rate=0.2*, *weight_decay=1.5e-05*, *input_height=32*, *batch_size=32*, *num_workers=0*, *warmup_epochs=10*, *max_epochs=1000*, ***kwargs*)

Bases: pytorch_lightning.LightningModule

PyTorch Lightning implementation of Bring 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:

```
import pytorch_lightning as pl
from pl_bolts.models.self_supervised import BYOL
from pl_bolts.datamodules import CIFAR10DataModule
from pl_bolts.models.self_supervised.simclr.simclr_transforms import (
    SimCLREvalDataTransform, SimCLRTrainDataTransform)

# model
model = BYOL(num_classes=10)

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

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

Train:

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

CLI command:

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

(continues on next page)

(continued from previous page)

```
# 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

22.2.3 CPC (V2)

class pl_bolts.models.self_supervised.**CPCV2**(*datamodule=None*, *encoder='cpc_encoder'*, *patch_size=8*, *patch_overlap=4*, *online_ft=True*, *task='cpc'*, *num_workers=4*, *learning_rate=0.0001*, *data_dir=''*, *batch_size=32*, *pretrained=None*, ***kwargs*)

Bases: pytorch_lightning.LightningModule

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

Example

```
>>> from pl_bolts.models.self_supervised import CPCV2
...
>>> model = CPCV2()
```

Train:

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

CLI command:

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

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

To Finetune:

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

Some uses:

```
# load resnet18 pretrained using CPC on imagenet
model = CPCV2(encoder='resnet18', pretrained=True)
resnet18 = model.encoder
renset18.freeze()

# it supportes any torchvision resnet
model = CPCV2(encoder='resnet50', pretrained=True)

# use it as a feature extractor
x = torch.rand(2, 3, 224, 224)
out = model(x)
```

Parameters

- **datamodule** (Optional[LightningDataModule]) – A Datamodule (optional). Otherwise set the dataloaders directly
- **encoder** (Union[str, Module, LightningModule]) – A string for any of the resnets in torchvision, or the original CPC encoder, or a custon nn.Module encoder
- **patch_size** (int) – How big to make the image patches
- **patch_overlap** (int) – How much overlap should each patch have.
- **online_ft** (int) – Enable a 1024-unit MLP to fine-tune online
- **task** (str) – Which self-supervised task to use ('cpc', 'amdim', etc...)
- **num_workers** (int) – num dataloader worksers

- **learning_rate** (int) – what learning rate to use
- **data_dir** (str) – where to store data
- **batch_size** (int) – batch size
- **pretrained** (Optional[str]) – If true, will use the weights pretrained (using CPC) on Imagenet

---

22.2.4 Moco (V2)

class pl_bolts.models.self_supervised.**MocoV2**(*base_encoder='resnet18'*, *emb_dim=128*, *num_negatives=65536*, *encoder_momentum=0.999*, *softmax_temperature=0.07*, *learning_rate=0.03*, *momentum=0.9*, *weight_decay=0.0001*, *datamodule=None*, *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()
```

Train:

```
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** (Optional[LightningDataModule]) – 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. ***

_momentum_update_key_encoder()
Momentum update of the key encoder

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

22.2.5 SimCLR

PyTorch Lightning implementation of SIMCLR

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

Model implemented by:

- William Falcon
- Tullie Murrell

Example:

```
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.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)

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

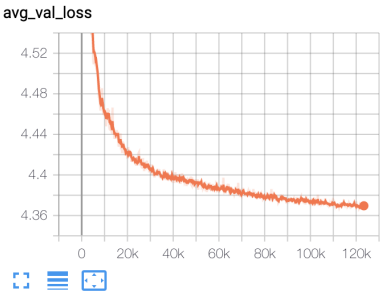
CIFAR-10 baseline

Table 1: Cifar-10 test accuracy

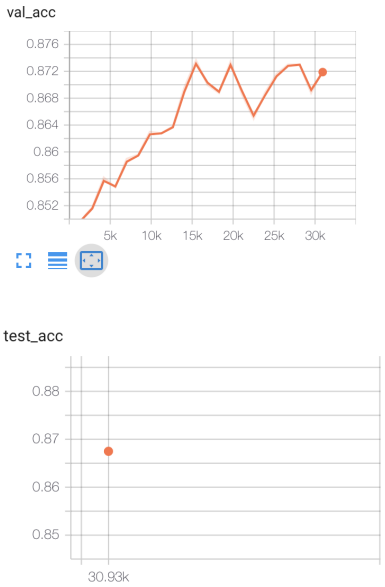
Model	test accuracy
Original repo	82.00
Our implementation	86.75

Note: This experiment used a standard resnet50 (not extra-wide, 2x, 4x). But you can use any resnet

Pre-training:



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



To reproduce:

```
# pretrain
python simclr_module.py
    --gpus 1
    --dataset cifar10
    --batch_size 512
    --learning_rate 1e-06
    --num_workers 8

# finetune
python simclr_finetuner.py
    --ckpt_path path/to/epoch=xyz.ckpt
    --gpus 1
```

SimCLR API

class pl_bolts.models.self_supervised.**SimCLR**(*batch_size*, *num_samples*, *warmup_epochs=10*, *lr=0.0001*, *opt_weight_decay=1e-06*, *loss_temperature=0.5*, ***kwargs*)

Bases: pytorch_lightning.LightningModule

Parameters

- **batch_size** – the batch size
- **num_samples** – num samples in the dataset
- **warmup_epochs** – epochs to warmup the lr for
- **lr** – the optimizer learning rate
- **opt_weight_decay** – the optimizer weight decay

- **loss_temperature** – the loss temperature

CHAPTER

TWENTYTHREE

SELF-SUPERVISED LEARNING TRANSFORMS

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

23.1 CPC transforms

Transforms used for CPC

23.1.1 CIFAR-10 Train (c)

class pl_bolts.models.self_supervised.cpc.transforms.**CPCTrainTransformsCIFAR10**(*patch_size=8*, *overlap=4*)

Bases: object

Transforms used for CPC:

Parameters

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

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())
```

__call__(*inp*)
Call self as a function.

23.1.2 CIFAR-10 Eval (c)

class pl_bolts.models.self_supervised.cpc.transforms.**CPCEvalTransformsCIFAR10**(*patch_size=8*, *overlap=4*)

Bases: object

Transforms used for CPC:

Parameters

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

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())
```

__call__(*inp*)
Call self as a function.

23.1.3 Imagenet Train (c)

class pl_bolts.models.self_supervised.cpc.transforms.**CPCTrainTransformsImageNet128**(*patch_size=*, *overlap=16*)

Bases: object

Transforms used for CPC:

Parameters

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

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())
```

__call__(*inp*)
Call self as a function.

23.1.4 Imagenet Eval (c)

class pl_bolts.models.self_supervised.cpc.transforms.**CPCEvalTransformsImageNet128**(*patch_size=*, *overlap=16*)

Bases: object

Transforms used for CPC:

Parameters

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

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())
```

__call__(*inp*)
Call self as a function.

23.1.5 STL-10 Train (c)

class pl_bolts.models.self_supervised.cpc.transforms.**CPCTrainTransformsSTL10**(*patch_size=16*, *overlap=8*)

Bases: object

Transforms used for CPC:

Parameters

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

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())
```

__call__(*inp*)

Call self as a function.

23.1.6 STL-10 Eval (c)

class pl_bolts.models.self_supervised.cpc.transforms.**CPCEvalTransformsSTL10**(*patch_size=16*, *overlap=8*)

Bases: object

Transforms used for CPC:

Parameters

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

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())
```

__call__(*inp*)
Call self as a function.

23.2 AMDIM transforms

Transforms used for AMDIM

23.2.1 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.

23.2.2 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.

23.2.3 Imagenet Train (a)

class pl_bolts.models.self_supervised.amdim.transforms.**AMDIMTrainTransformsImageNet128**(*heigh*
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.

23.2.4 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.

23.2.5 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.

23.2.6 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.

23.3 MOCO V2 transforms

Transforms used for MOCO V2

23.3.1 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.

23.3.2 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.

23.3.3 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.

23.3.4 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.

23.3.5 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.

23.3.6 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.

23.4 SimCLR transforms

Transforms used for SimCLR

23.4.1 Train (sc)

class pl_bolts.models.self_supervised.simclr.simclr_transforms.**SimCLRTrainDataTransform**(*inp*
s=1

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.

23.4.2 Eval (sc)

class pl_bolts.models.self_supervised.simclr.simclr_transforms.**SimCLREvalDataTransform**(*input_height*, *s=1*)

Bases: object

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)
```

__call__(*sample*)

Call self as a function.

CHAPTER

TWENTYFOUR

SELF-SUPERVISED LEARNING

Collection of useful functions for self-supervised learning

24.1 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()
```

24.2 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*, *return_all_feature_maps=False*)

24.3 SSL backbone finetuner

class pl_bolts.models.self_supervised.ssl_finetuner.**SSLFineTuner**(*backbone*, *in_features*, *num_classes*, *hidden_dim=1024*)

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** – a pretrained model
- **in_features** – feature dim of backbone outputs
- **num_classes** – classes of the dataset
- **hidden_dim** – dim of the MLP (1024 default used in self-supervised literature)

CHAPTER

TWENTYFIVE

SEMI-SUPERVISED LEARNING

Collection of utilities for semi-supervised learning where some part of the data is labeled and the other part is not.

25.1 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** (ndarray) – input features
- **Y** (list) – mixed labels (ints)
- **batch_size** (int) – the ultimate batch size

25.2 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

CHAPTER

TWENTYSIX

SELF-SUPERVISED LEARNING CONTRASTIVE TASKS

This section implements popular contrastive learning tasks used in self-supervised learning.

26.1 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 BxB 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*, *bidirectional=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))
```

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

```
# 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)
```

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)]
```

26.2 Context prediction tasks

The following tasks aim to predict a target using a context representation.

26.2.1 CPCContrastiveTask

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*, *embed_scale=0.1*)

Bases: torch.nn.Module

Loss used in CPC

CHAPTER

TWENTYSEVEN

INDICES AND TABLES

- genindex
- modindex
- search

Logo

27.1 PyTorch Lightning Bolts

Pretrained SOTA Deep Learning models, callbacks and more for research and production with PyTorch Lightning and PyTorch

---

27.1.1 Trending contributors

27.1.2 Continuous Integration

| System / PyTorch ver. | 1.4 (min. req.) | 1.5 (latest) | | :—: | :—: | :—: | | Linux py3.6 / py3.7 / py3.8 | CI testing | CI testing | | OSX py3.6 / py3.7 / py3.8 | CI testing | CI testing | | Windows py3.6 / py3.7 / py3.8 | wip | wip |

27.1.3 Install

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

27.1.4 Docs

- master
- stable
- 0.1.0

27.1.5 What is Bolts

Bolts is a Deep learning research and production toolbox of:

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

27.1.6 Main Goals of Bolts

The main goal of Bolts is to enable rapid model idea iteration.

Example 1: Finetuning on data

```
from pl_bolts.models.self_supervised import SimCLR
from pl_bolts.models.self_supervised.simclr.transforms import␣
↪SimCLRTrainDataTransform, SimCLREvalDataTransform
import pytorch_lightning as pl

# data
train_data = DataLoader(MyDataset(transforms=SimCLRTrainDataTransform(input_
↪height=32)))
val_data = DataLoader(MyDataset(transforms=SimCLREvalDataTransform(input_height=32)))

# model
model = SimCLR(pretrained='imagenet2012')

# train!
trainer = pl.Trainer(gpus=8)
trainer.fit(model, train_data, val_data)
```

Example 2: Subclass and ideate

```
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
        # ------------------
```

(continues on next page)

(continued from previous page)

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

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

27.1.7 Who is Bolts for?

- Corporate production teams
- Professional researchers
- Ph.D. students
- Linear + Logistic regression heroes

27.1.8 I don't need deep learning

Great! We have LinearRegression and LogisticRegression implementations with numpy and sklearn bridges for datasets! But our implementations work on multiple GPUs, TPUs and scale dramatically...

Check out our Linear Regression on TPU demo

```
from pl_bolts.models.regression import LinearRegression
from pl_bolts.datamodules import SklearnDataModule

# sklearn dataset
X, y = load_boston(return_X_y=True)
loaders = SklearnDataModule(X, y)

model = LinearRegression(input_dim=13)
trainer = pl.Trainer(num_tpu_cores=1)
trainer.fit(model, loaders.train_dataloader(), loaders.val_dataloader())
trainer.test(test_dataloaders=loaders.test_dataloader())
```

27.1.9 Is this another model zoo?

No!

Bolts is unique because models are implemented using PyTorch Lightning and structured so that they can be easily subclassed and iterated on.

For example, you can override the elbo loss of a VAE, or the generator_step of a GAN to quickly try out a new idea. The best part is that all the models are benchmarked so you won't waste time trying to "reproduce" or find the bugs with your implementation.

27.1.10 Team

Bolts is supported by the PyTorch Lightning team and the PyTorch Lightning community!

27.2 pl_bolts.callbacks package

Collection of PyTorchLightning callbacks

27.2.1 Subpackages

pl_bolts.callbacks.vision package

Submodules

pl_bolts.callbacks.vision.confused_logit module

class pl_bolts.callbacks.vision.confused_logit.**ConfusedLogitCallback**(*top_k*, *projection_factor=3*, *min_logit_value=5.0*, *logging_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

Parameters

- **top_k** – How many "offending" images we should plot

- **projection_factor** – How much to multiply the input image to make it look more like this logit label
- **min_logit_value** – Only consider logit values above this threshold
- **logging_batch_interval** – how frequently to inspect/potentially plot something
- **max_logit_difference** – when the top 2 logits are within this threshold we consider them confused

Authored by:

- Alfredo Canziani

static _ConfusedLogitCallback__draw_sample(*fig*, *axarr*, *row_idx*, *col_idx*, *img*, *title*)

_plot(*confusing_x*, *confusing_y*, *trainer*, *model*, *mask_idxs*)

on_train_batch_end(*trainer*, *pl_module*, *batch*, *batch_idx*, *dataloader_idx*)

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()])
```

on_epoch_end(*trainer*, *pl_module*)

27.2.2 Submodules

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
# ------------------------------

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

on_epoch_end(*trainer*, *pl_module*)

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]]) – formating 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
```

pl_bolts.callbacks.self_supervised module

class pl_bolts.callbacks.self_supervised.**BYOLMAWeightUpdate**(*initial_tau=0.996*)
Bases: pytorch_lightning.Callback

Weight update rule from BYOL.

Your model should have a:

- 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:

```
from pl_bolts.callbacks.self_supervised import BYOLMAWeightUpdate

# model must have 2 attributes
model = Model()
model.online_network = ...
model.target_network = ...

# make sure to set max_steps in Trainer
trainer = Trainer(callbacks=[BYOLMAWeightUpdate()], max_steps=1000)
```

Parameters **initial_tau** – starting tau. Auto-updates with every training step

on_train_batch_end(*trainer*, *pl_module*, *batch*, *batch_idx*, *dataloader_idx*)

update_tau(*pl_module*, *trainer*)

update_weights(*online_net*, *target_net*)

class pl_bolts.callbacks.self_supervised.**SSLOnlineEvaluator**(*drop_p=0.2*, *hidden_dim=1024*, *z_dim=None*, *num_classes=None*)
Bases: pytorch_lightning.Callback

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

Example:

```
from pl_bolts.callbacks.self_supervised import SSLOnlineEvaluator

# your model must have 2 attributes
model = Model()
model.z_dim = ... # the representation dim
model.num_classes = ... # the num of classes in the model
```

Parameters

- **drop_p** (float) – (0.2) dropout probability
- **hidden_dim** (int) –

(1024) the hidden dimension for the finetune MLP

get_representations(*pl_module*, *x*)
Override this to customize for the particular model :param _sphinx_paramlinks_pl_bolts.callbacks.self_supervised.SSLOnlin
:param _sphinx_paramlinks_pl_bolts.callbacks.self_supervised.SSLOnlineEvaluator.get_representations.x:

on_pretrain_routine_start(*trainer*, *pl_module*)

on_train_batch_end(*trainer*, *pl_module*, *batch*, *batch_idx*, *dataloader_idx*)

to_device(*batch*, *device*)

pl_bolts.callbacks.variational module

class pl_bolts.callbacks.variational.**LatentDimInterpolator**(*interpolate_epoch_interval=20*, *range_start=-5*, *range_end=5*, *num_samples=2*)

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** –
- **range_start** – default -5
- **range_end** – default 5
- **num_samples** – default 2

interpolate_latent_space(*pl_module*, *latent_dim*)

on_epoch_end(*trainer*, *pl_module*)

27.3 pl_bolts.datamodules package

27.3.1 Submodules

pl_bolts.datamodules.async_dataloader module

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** – The PyTorch Dataset or DataLoader we're using to load.
- **device** – The PyTorch device we are loading to
- **q_size** – Size of the queue used to store the data loaded to the device
- **num_batches** – 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*)

load_loop()

pl_bolts.datamodules.base_dataset module

class pl_bolts.datamodules.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[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

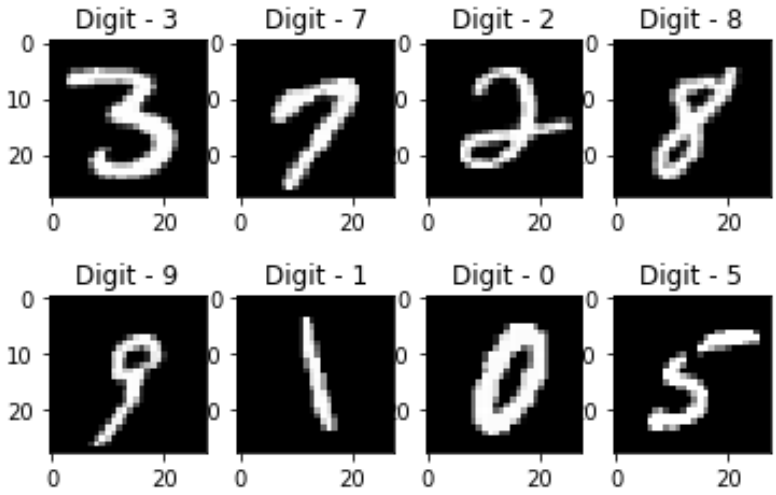
pl_bolts.datamodules.binary_mnist_datamodule module

class pl_bolts.datamodules.binary_mnist_datamodule.**BinaryMNIST**(**args*, ***kwargs*)

Bases: torchvision.datasets.MNIST

class pl_bolts.datamodules.binary_mnist_datamodule.**BinaryMNISTDataModule**(*data_dir*, *val_split=5000*, *num_workers=16*, *normalize=False*, *seed=42*, **args*, ***kwargs*)

Bases: pytorch_lightning.LightningDataModule



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, dm)
```

Parameters

- **data_dir** (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
- **normalize** (bool) – If true applies image normalize

_default_transforms()

prepare_data()
Saves MNIST files to data_dir

test_dataloader(*batch_size=32*, *transforms=None*)
MNIST test set uses the test split

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

train_dataloader(*batch_size=32*, *transforms=None*)
MNIST train set removes a subset to use for validation

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

val_dataloader(*batch_size=32*, *transforms=None*)
MNIST val set uses a subset of the training set for validation

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

name = 'mnist'

property num_classes
Return: 10

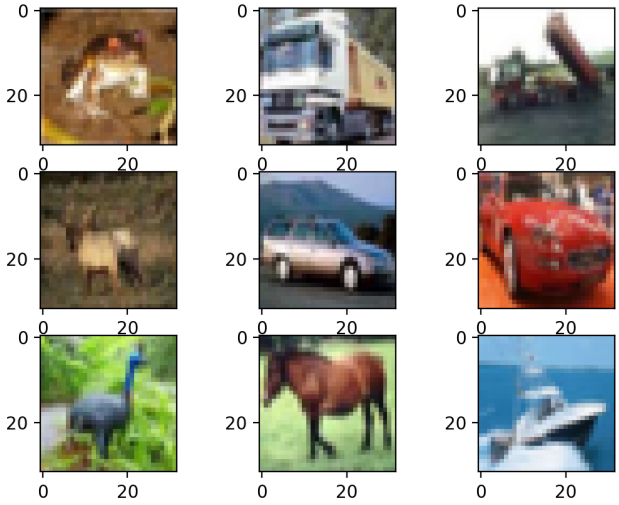
pl_bolts.datamodules.cifar10_datamodule module

class pl_bolts.datamodules.cifar10_datamodule.**CIFAR10DataModule**(*data_dir=None*, *val_split=5000*, *num_workers=16*, *batch_size=32*, *seed=42*, **args*, ***kwargs*)

Bases: pytorch_lightning.LightningDataModule

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, 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** (int) – how many of the training images to use for the validation split
- **num_workers** (int) – how many workers to use for loading data
- **batch_size** (int) – number of examples per training/eval step

default_transforms()

prepare_data()
Saves CIFAR10 files to data_dir

test_dataloader()
CIFAR10 test set uses the test split

train_dataloader()
CIFAR train set removes a subset to use for validation

val_dataloader()
CIFAR10 val set uses a subset of the training set for validation

extra_args = {}

name = 'cifar10'

property num_classes
Return: 10

class pl_bolts.datamodules.cifar10_datamodule.**TinyCIFAR10DataModule**(*data_dir*, *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** (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

property num_classes
Return number of classes.

Return type int

pl_bolts.datamodules.cifar10_dataset module

class pl_bolts.datamodules.cifar10_dataset.**CIFAR10**(*data_dir='.'*, *train=True*, *transform=None*, *download=True*)

Bases: *pl_bolts.datamodules.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)
>>> 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.datamodules.cifar10_dataset.**TrialCIFAR10**(*data_dir='.'*, *train=True*, *transform=None*, *download=False*, *num_samples=100*, *labels=(1, 5, 8)*, *relabel=True*)

Bases: *pl_bolts.datamodules.cifar10_dataset.CIFAR10*

Customized CIFAR10 dataset for testing Pytorch Lightning without the torchvision dependency.

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

Examples

```
>>> dataset = TrialCIFAR10(download=True, num_samples=150, labels=(1, 5, 8))
>>> 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])
```

prepare_data(*download*)

Return type None

data = None

dir_path = None

normalize = None

targets = None

pl_bolts.datamodules.cityscapes_datamodule module

class pl_bolts.datamodules.cityscapes_datamodule.**CityscapesDataModule**(*data_dir*, *val_split=5000*, *num_workers=16*, *batch_size=32*, *seed=42*, **args*, ***kwargs*)

Bases: pytorch_lightning.LightningDataModule

Standard Cityscapes, train, val, test splits and transforms

Specs:

- 30 classes (road, person, sidewalk, etc...)
- (image, target) - image dims: (3 x 32 x 32), target dims: (3 x 32 x 32)

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, dm)
```

Or you can set your own transforms

Example:

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

Parameters

- **data_dir** – where to save/load the data
- **val_split** – how many of the training images to use for the validation split
- **num_workers** – how many workers to use for loading data
- **batch_size** – number of examples per training/eval step

default_transforms()

prepare_data()
Saves Cityscapes files to data_dir

test_dataloader()
Cityscapes test set uses the test split

train_dataloader()
Cityscapes train set with removed subset to use for validation

val_dataloader()
Cityscapes val set uses a subset of the training set for validation

extra_args = {}

name = 'Cityscapes'

property num_classes
Return: 30

pl_bolts.datamodules.concat_dataset module

class pl_bolts.datamodules.concat_dataset.**ConcatDataset**(**datasets*)
Bases: torch.utils.data.Dataset

pl_bolts.datamodules.dummy_dataset module

class pl_bolts.datamodules.dummy_dataset.**DummyDataset**(**shapes*, *num_samples=10000*)
Bases: torch.utils.data.Dataset

Generate a dummy dataset

Parameters

- ***shapes** – list of shapes
- **num_samples** – how many samples to use in this dataset

Example:

```
from pl_bolts.datamodules import DummyDataset

# mnist dims
>>> ds = DummyDataset((1, 28, 28), (1,))
>>> dl = DataLoader(ds, batch_size=7)
...
>>> batch = next(iter(dl))
>>> x, y = batch
>>> x.size()
torch.Size([7, 1, 28, 28])
>>> y.size()
torch.Size([7, 1])
```

class pl_bolts.datamodules.dummy_dataset.**DummyDetectionDataset**(*img_shape=(3, 256, 256)*, *num_boxes=1*, *num_classes=2*, *num_samples=10000*)
Bases: torch.utils.data.Dataset

_random_bbox()

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

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(***kwds*)
Return a new Experience object replacing specified fields with new values

_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

Return type 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** (Env) – 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

Return type 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 lastest 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. :param _sphinx_paramlinks_pl_bolts.datamodules.experience_source.ExperienceSource.update_history_queue.env_idx: index of the environment :param _sphinx_paramlinks_pl_bolts.datamodules.experience_source.ExperienceSource.update_hi the current experience :param _sphinx_paramlinks_pl_bolts.datamodules.experience_source.ExperienceSource.update_histo 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*, *val_split=5000*, *num_workers=16*, *seed=42*, **args*, ***kwargs*)
Bases: pytorch_lightning.LightningDataModule

Specs:

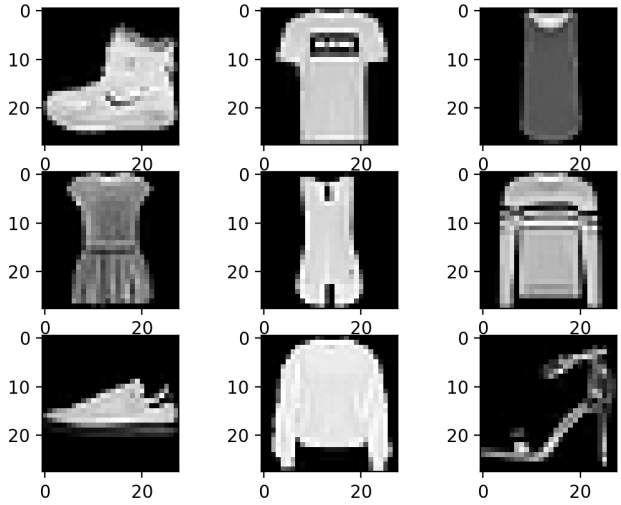
- 10 classes (1 per type)
- Each image is (1 x 28 x 28)

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, dm)
```

Parameters

- **data_dir** (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

_default_transforms()

prepare_data()
Saves FashionMNIST files to data_dir

test_dataloader(*batch_size=32*, *transforms=None*)
FashionMNIST test set uses the test split

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

train_dataloader(*batch_size=32*, *transforms=None*)
FashionMNIST train set removes a subset to use for validation

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

val_dataloader(*batch_size=32*, *transforms=None*)
FashionMNIST val set uses a subset of the training set for validation

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

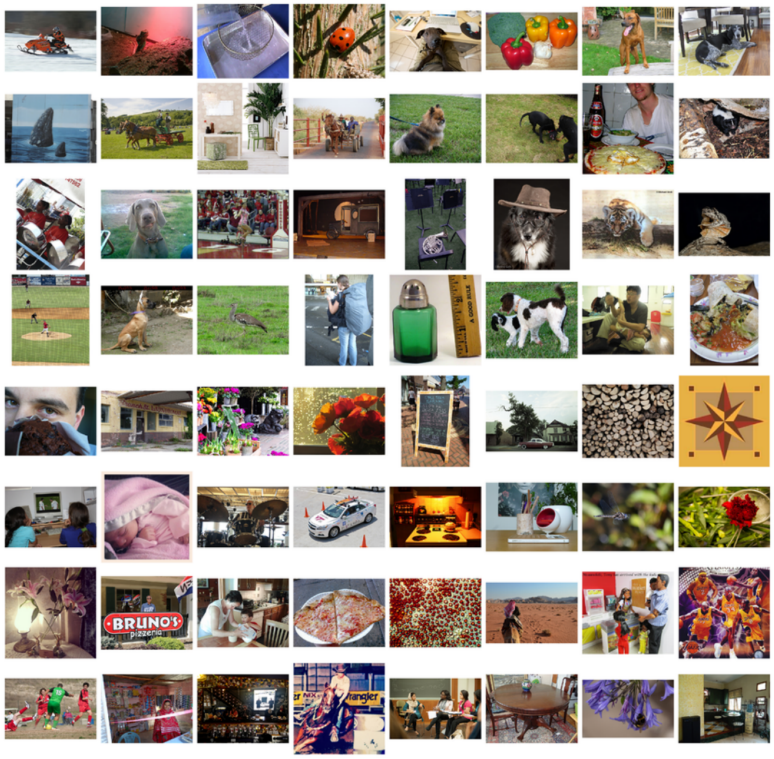
name = 'fashion_mnist'

property num_classes
Return: 10

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*, **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(IMAGENET_PATH)
model = LitModel()

Trainer().fit(model, 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

_verify_splits(*data_dir*, *split*)

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.

test_dataloader()
Uses the validation split of imagenet2012 for testing

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

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]
    ),
])
```

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

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]
    ),
])
```

name = 'imagenet'

property num_classes
Return:

1000

pl_bolts.datamodules.imagenet_dataset module

class pl_bolts.datamodules.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: torchvision.datasets.ImageNet

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.datamodules.imagenet_dataset.**_calculate_md5**(*fpath*, *chunk_size=1048576*)

pl_bolts.datamodules.imagenet_dataset.**_check_integrity**(*fpath*, *md5=None*)

pl_bolts.datamodules.imagenet_dataset.**_check_md5**(*fpath*, *md5*, ***kwargs*)

pl_bolts.datamodules.imagenet_dataset.**_is_gzip**(*filename*)

pl_bolts.datamodules.imagenet_dataset.**_is_tar**(*filename*)

pl_bolts.datamodules.imagenet_dataset.**_is_targz**(*filename*)

pl_bolts.datamodules.imagenet_dataset.**_is_tarxz**(*filename*)

pl_bolts.datamodules.imagenet_dataset.**_is_zip**(*filename*)

pl_bolts.datamodules.imagenet_dataset.**_verify_archive**(*root*, *file*, *md5*)

pl_bolts.datamodules.imagenet_dataset.**extract_archive**(*from_path*, *to_path=None*, *remove_finished=False*)

pl_bolts.datamodules.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.

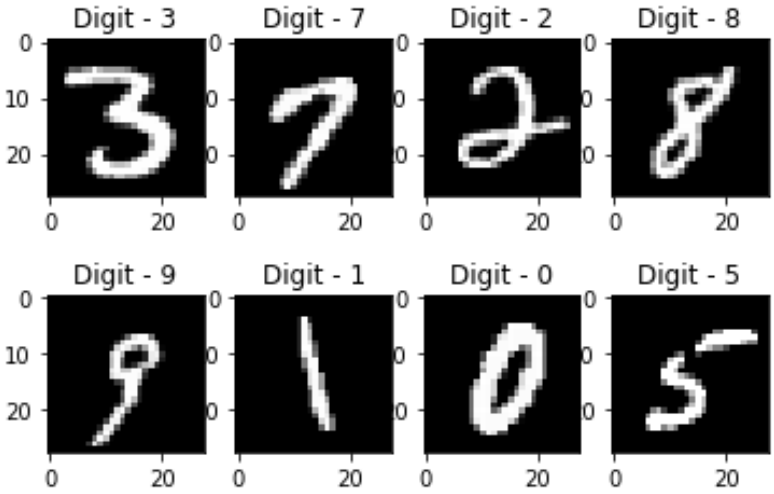
Parameters

- **root** (*str*) – Root directory containing the devkit archive
- **file** (*str, optional*) – Name of devkit archive. Defaults to 'ILSVRC2012_devkit_t12.tar.gz'

pl_bolts.datamodules.mnist_datamodule module

class pl_bolts.datamodules.mnist_datamodule.**MNISTDataModule**(*data_dir*, *val_split=5000*, *num_workers=16*, *normalize=False*, *seed=42*, **args*, ***kwargs*)

Bases: pytorch_lightning.LightningDataModule



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, dm)
```

Parameters

- **data_dir** (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
- **normalize** (bool) – If true applies image normalize

_default_transforms()

prepare_data()
Saves MNIST files to data_dir

test_dataloader(*batch_size=32*, *transforms=None*)
MNIST test set uses the test split

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

train_dataloader(*batch_size=32*, *transforms=None*)
MNIST train set removes a subset to use for validation

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

val_dataloader(*batch_size=32*, *transforms=None*)
MNIST val set uses a subset of the training set for validation

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

name = 'mnist'

property num_classes
Return: 10

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*, **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)
...
>>> # train set
>>> train_loader = loaders.train_dataloader(batch_size=32)
>>> len(train_loader.dataset)
355
>>> len(train_loader)
11
>>> # validation set
>>> val_loader = loaders.val_dataloader(batch_size=32)
>>> len(val_loader.dataset)
100
>>> len(val_loader)
3
>>> # test set
>>> test_loader = loaders.test_dataloader(batch_size=32)
>>> len(test_loader.dataset)
51
>>> len(test_loader)
1
```

_init_datasets(*X*, *y*, *x_val*, *y_val*, *x_test*, *y_test*)

test_dataloader(*batch_size=16*)

train_dataloader(*batch_size=16*)

val_dataloader(*batch_size=16*)

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.

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

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
```

class pl_bolts.datamodules.sklearn_datamodule.**TensorDataModule**(*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*, **args*, ***kwargs*)

Bases: *pl_bolts.datamodules.sklearn_datamodule.SklearnDataModule*

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

Example

```
>>> from pl_bolts.datamodules import TensorDataModule
>>> import torch
...
>>> # create dataset
>>> X = torch.rand(100, 3)
>>> y = torch.rand(100)
>>> loaders = TensorDataModule(X, y)
...
>>> # train set
>>> train_loader = loaders.train_dataloader(batch_size=10)
>>> len(train_loader.dataset)
70
>>> len(train_loader)
7
>>> # validation set
>>> val_loader = loaders.val_dataloader(batch_size=10)
>>> len(val_loader.dataset)
20
>>> len(val_loader)
2
>>> # test set
>>> test_loader = loaders.test_dataloader(batch_size=10)
>>> len(test_loader.dataset)
10
>>> len(test_loader)
1
```

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)
...
>>> # train set
>>> train_loader = loaders.train_dataloader(batch_size=32)
>>> len(train_loader.dataset)
355
>>> len(train_loader)
11
>>> # validation set
>>> val_loader = loaders.val_dataloader(batch_size=32)
>>> len(val_loader.dataset)
100
>>> len(val_loader)
3
>>> # test set
>>> test_loader = loaders.test_dataloader(batch_size=32)
>>> len(test_loader.dataset)
51
```

(continues on next page)

(continued from previous page)

```
>>> len(test_loader)
1
```

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.

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

Example

```
>>> from pl_bolts.datamodules import TensorDataset
...
>>> X = torch.rand(10, 3)
>>> y = torch.rand(10)
>>> dataset = TensorDataset(X, y)
>>> len(dataset)
10
```

pl_bolts.datamodules.ssl_imagenet_datamodule module

class pl_bolts.datamodules.ssl_imagenet_datamodule.**SSLImagenetDataModule**(*data_dir*, *meta_dir=None*, *num_workers=16*, **args*, ***kwargs*)

Bases: pytorch_lightning.LightningDataModule

_default_transforms()

_verify_splits(*data_dir*, *split*)

prepare_data()

test_dataloader(*batch_size*, *num_images_per_class*, *add_normalize=False*)

train_dataloader(*batch_size*, *num_images_per_class=-1*, *add_normalize=False*)

val_dataloader(*batch_size*, *num_images_per_class=50*, *add_normalize=False*)

name = 'imagenet'

property num_classes

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*, **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, 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

default_transforms()

prepare_data()
Downloads the unlabeled, train and test split

test_dataloader()
Loads the test split of STL10

Parameters

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

train_dataloader()
Loads the 'unlabeled' split minus a portion set aside for validation via *unlabeled_val_split*.

train_dataloader_labeled()

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

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

val_dataloader_labeled()

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

name = 'stl10'

property num_classes

pl_bolts.datamodules.vocdetection_datamodule module

class pl_bolts.datamodules.vocdetection_datamodule.**Compose**(*transforms*)
Bases: object

Like *torchvision.transforms.compose* but works for (image, target)

__call__(*image*, *target*)
Call self as a function.

class pl_bolts.datamodules.vocdetection_datamodule.**VOCDetectionDataModule**(*data_dir*, *year='2012'*, *num_workers=16*, *normalize=False*, **args*, ***kwargs*)
Bases: pytorch_lightning.LightningDataModule

TODO(teddykoker) docstring

_default_transforms()

prepare_data()
Saves VOCDetection files to data_dir

train_dataloader(*batch_size=1*, *transforms=None*)
VOCDetection train set uses the *train* subset

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

val_dataloader(*batch_size=1*, *transforms=None*)
VOCDetection val set uses the *val* subset

Parameters

- **batch_size** – size of batch
- **transforms** – custom transforms

name = 'vocdetection'

property num_classes
Return: 21

pl_bolts.datamodules.vocdetection_datamodule.**_collate_fn**(*batch*)

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

27.4 pl_bolts.metrics package

27.4.1 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

27.5 pl_bolts.models package

Collection of PyTorchLightning models

27.5.1 Subpackages

pl_bolts.models.autoencoders package

Here are a VAE and GAN

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 MNIST 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 and decoder are both convolutional with a 128-dimensional hidden layer and a 32-dimensional latent space. The model accepts arguments for these dimensions (see example below) if you want to use the default encoder + decoder but with different hidden layer and latent layer dimensions. The model also assumes a Gaussian prior and a Gaussian approximate posterior distribution.

```
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**(*datamodule=None*, *input_channels=1*, *input_height=28*, *input_width=28*, *latent_dim=32*, *batch_size=32*, *hidden_dim=128*, *learning_rate=0.001*, *num_workers=8*, *data_dir='.'*, ***kwargs*)

Bases: pytorch_lightning.LightningModule

Arg:

datamodule: the datamodule (train, val, test splits) input_channels: num of image channels input_height: image height input_width: image width latent_dim: emb dim for encoder batch_size: the batch size hidden_dim: the encoder dim learning_rate: the learning rate num_workers: num dataloader workers data_dir: where to store data

_run_step(*batch*)

static add_model_specific_args(*parent_parser*)

configure_optimizers()

forward(*z*)

init_decoder(*hidden_dim*, *latent_dim*)

init_encoder(*hidden_dim*, *latent_dim*, *input_width*, *input_height*)

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.autoencoders.basic_ae.components module

class pl_bolts.models.autoencoders.basic_ae.components.**AEEncoder**(*hidden_dim*, *latent_dim*, *input_width*, *input_height*)

Bases: torch.nn.Module

Takes as input an image, uses a CNN to extract features which get split into a mu and sigma vector

_calculate_output_dim(*input_width*, *input_height*)

forward(*x*)

class pl_bolts.models.autoencoders.basic_ae.components.**DenseBlock**(*in_dim*, *out_dim*, *drop_p=0.2*)

Bases: torch.nn.Module

forward(*x*)

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 MNIST 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 and decoder are both convolutional with a 128-dimensional hidden layer and a 32-dimensional latent space. The model accepts arguments for these dimensions (see example below) if you want to use the default encoder + decoder but with different hidden layer and latent layer dimensions. 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**(*hidden_dim=128*, *latent_dim=32*, *input_channels=3*, *input_width=224*, *input_height=224*, *batch_size=32*, *learning_rate=0.001*, *data_dir='.'*, *datamodule=None*, *num_workers=8*, *pretrained=None*, ***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 imagenet
vae = VAE(pretrained='imagenet')

# pretrained on cifar10
vae = VAE(pretrained='cifar10')
```

Parameters

- **hidden_dim** (int) – encoder and decoder hidden dims
- **latent_dim** (int) – latenet code dim
- **input_channels** (int) – num of channels of the input image.
- **input_width** (int) – image input width
- **input_height** (int) – image input height
- **batch_size** (int) – the batch size
- **the learning rate** (*learning_rate"*) –
- **data_dir** (str) – the directory to store data
- **datamodule** (Optional[LightningDataModule]) – The Lightning DataModule

- **pretrained** (Optional[str]) – Load weights pretrained on a dataset

_VAE__init_system()

_VAE__set_pretrained_dims(*pretrained*)

_run_step(*batch*)

_set_default_datamodule(*datamodule*)

static add_model_specific_args(*parent_parser*)

configure_optimizers()

elbo_loss(*x*, *P*, *Q*, *num_samples*)

forward(*z*)

get_approx_posterior(*z_mu*, *z_std*)

get_prior(*z_mu*, *z_std*)

init_decoder()

init_encoder()

load_pretrained(*pretrained*)

test_step(*batch*, *batch_idx*)

training_step(*batch*, *batch_idx*)

validation_step(*batch*, *batch_idx*)

pl_bolts.models.autoencoders.basic_vae.basic_vae_module.**cli_main**()

pl_bolts.models.autoencoders.basic_vae.components module

class pl_bolts.models.autoencoders.basic_vae.components.**Decoder**(*hidden_dim*, *latent_dim*, *input_width*, *input_height*, *input_channels*)

Bases: torch.nn.Module

Takes in latent vars and reconstructs an image

_calculate_output_size(*input_width*, *input_height*)

forward(*z*)

class pl_bolts.models.autoencoders.basic_vae.components.**DenseBlock**(*in_dim*, *out_dim*, *drop_p=0.2*)

Bases: torch.nn.Module

forward(*x*)

class pl_bolts.models.autoencoders.basic_vae.components.**Encoder**(*hidden_dim*, *latent_dim*, *input_channels*, *input_width*, *input_height*)

Bases: torch.nn.Module

Takes as input an image, uses a CNN to extract features which get split into a mu and sigma vector

_calculate_output_dim(*input_width*, *input_height*)

forward(*x*)

pl_bolts.models.detection package

Submodules

pl_bolts.models.detection.faster_rcnn module

class pl_bolts.models.detection.faster_rcnn.**FasterRCNN**(*learning_rate=0.0001*, *num_classes=91*, *pretrained=False*, *pretrained_backbone=True*, *trainable_backbone_layers=3*, *replace_head=True*, ***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 truh 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)
- **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.**_evaluate_iou**(*target*, *pred*)
Evaluate intersection over union (IOU) for target from dataset and output prediction from model

pl_bolts.models.detection.faster_rcnn.**run_cli**()

pl_bolts.models.gans package

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**(*datamodule=None*, *latent_dim=32*, *batch_size=100*, *learning_rate=0.0002*, *data_dir=''*, *num_workers=8*, ***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

- **datamodule** (Optional[LightningDataModule]) – the datamodule (train, val, test splits)
- **latent_dim** (int) – emb dim for encoder
- **batch_size** (int) – the batch size

- **learning_rate** (float) – the learning rate
- **data_dir** (str) – where to store data
- **num_workers** (int) – data workers

_set_default_datamodule(*datamodule*)

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**()

pl_bolts.models.gans.basic.components module

class pl_bolts.models.gans.basic.components.**Discriminator**(*img_shape*, *hidden_dim=1024*)

Bases: torch.nn.Module

forward(*img*)

class pl_bolts.models.gans.basic.components.**Generator**(*latent_dim*, *img_shape*, *hidden_dim=256*)

Bases: torch.nn.Module

forward(*z*)

pl_bolts.models.regression package

Submodules

pl_bolts.models.regression.linear_regression module

class pl_bolts.models.regression.linear_regression.**LinearRegression**(*input_dim*, *bias=True*, *learning_rate=0.0001*, *optimizer=torch.optim.Adam*, *l1_strength=None*, *l2_strength=None*, ***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+)
- **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** (Optional[float]) – L1 regularization strength (default=None)
- **l2_strength** (Optional[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 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.rl package

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 :type _sphinx_paramlinks_pl_bolts.models.rl.common.agents.ValueAgent.get_action.state: Tensor :param _sphinx_paramlinks_pl_bolts.models.rl.common.agents.ValueAgent.get_action.state: current state of the environment :type _sphinx_paramlinks_pl_bolts.models.rl.common.agents.ValueAgent.get_action.device: device :param _sphinx_paramlinks_pl_bolts.models.rl.common.agents.ValueAgent.get_action.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.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

Return type 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(***kwds*)
Return a new Experience object replacing specified fields with new values

_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**(*buffer_size*, *n_step=2*)
Bases: object

N Step Replay Buffer

Deprecated: use the NStepExperienceSource with the standard ReplayBuffer

append(*experience*)
add an experience to the buffer by collecting n steps of experiences :param _sphinx_paramlinks_pl_bolts.models.rl.common.memory.MultiStepBuffer.append.experience: tuple (state, action, reward, done, next_state)

Return type None

get_transition_info(*gamma=0.9*)
get the accumulated transition info for the n_step_buffer :param _sphinx_paramlinks_pl_bolts.models.rl.common.memory.MultiStepBuffer.get_transition_info.gamma: discount factor

Return type Tuple[float, array, int]

Returns multi step reward, final observation and done

sample(*batch_size*)
Takes a sample of the buffer :type _sphinx_paramlinks_pl_bolts.models.rl.common.memory.MultiStepBuffer.sample.batch_s
int :param _sphinx_paramlinks_pl_bolts.models.rl.common.memory.MultiStepBuffer.sample.batch_size: current batch_size

Return type Tuple

Returns a batch of tuple np arrays of Experiences

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

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 :type _sphinx_paramlinks_pl_bolts.models.rl.common.memory.ReplayBuffer.sample.batch_size int :param _sphinx_paramlinks_pl_bolts.models.rl.common.memory.ReplayBuffer.sample.batch_size: 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** – observation shape of the environment

- **n_actions** – 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) – 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) – 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) – 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** – observation shape of the environment
- **n_actions** – 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** – number of inputs
- **out_features** – number of outputs
- **sigma_init** – initial fill value of noisy weights
- **bias** – 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 paramseter of the layer

Return type None

pl_bolts.models.rl.common.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.wrappers.**BufferWrapper**(*env*, *n_steps*, *dtype=numpy.float32*)

Bases: gym.ObservationWrapper

"Wrapper for image stacking

observation(*observation*)
convert observation

reset()
reset env

class pl_bolts.models.rl.common.wrappers.**DataAugmentation**(*env=None*)
Bases: gym.ObservationWrapper

Carries out basic data augmentation on the env observations

- ToTensor
- GrayScale
- RandomCrop

observation(*obs*)
preprocess the obs

class pl_bolts.models.rl.common.wrappers.**FireResetEnv**(*env=None*)
Bases: gym.Wrapper

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.wrappers.**ImageToPyTorch**(*env*)
Bases: gym.ObservationWrapper

converts image to pytorch format

static observation(*observation*)
convert observation

class pl_bolts.models.rl.common.wrappers.**MaxAndSkipEnv**(*env=None*, *skip=4*)
Bases: gym.Wrapper

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.wrappers.**ProcessFrame84**(*env=None*)
Bases: gym.ObservationWrapper

preprocessing images from env

observation(*obs*)
preprocess the obs

static process(*frame*)
image preprocessing, formats to 84x84

class pl_bolts.models.rl.common.wrappers.**ScaledFloatFrame**(**args*, ***kwargs*)
Bases: gym.ObservationWrapper

scales the pixels

static observation(*obs*)

class pl_bolts.models.rl.common.wrappers.**ToTensor**(*env=None*)
Bases: gym.Wrapper

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.wrappers.**make_environment**(*env_name*)
Convert environment with wrappers

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*, *replay_size=100000*, *warm_start_size=10000*, *avg_reward_len=100*, *min_episode_reward=-21*, *n_steps=1*, *seed=123*, *num_envs=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)
```

Parameters

- **env** (str) – gym environment tag
- **gpus** – number of gpus being used
- **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 espilon = eps_end
- **sync_rate** (int) – the number of iterations between syncing up the target network with the train network
- **gamma** (float) – discount factor
- **lr** – 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
- **sample_len** – the number of samples to pull from the dataset iterator and feed to the DataLoader

---

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*

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)
```

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 espilon = 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
- **num_envs** (int) – number of environments to run the agent in at once

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*

training_step(*batch*, _)

Carries out a single step through the environment to update the replay buffer. Then calculates loss based on the minibatch recieved :type _sphinx_paramlinks_pl_bolts.models.rl.double_dqn_model.DoubleDQN.training_step.batch: Tuple[Tensor, Tensor] :param _sphinx_paramlinks_pl_bolts.models.rl.double_dqn_model.DoubleDQN.training_step.b current mini batch of replay data :param _sphinx_paramlinks_pl_bolts.models.rl.double_dqn_model.DoubleDQN.training_st batch number, not used

Return type OrderedDict

Returns Training loss and log metrics

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*, *n_steps=1*, *seed=123*, *num_envs=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)
```

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 espilon = 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
- **num_envs** (int) – number of environments to run the agent in at once

---

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*

_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 :type _sphinx_paramlinks_pl_bolts.models.rl.dqn_model.DQN.add_model_specific_args.arg_parser:

ArgumentParser :param _sphinx_paramlinks_pl_bolts.models.rl.dqn_model.DQN.add_model_specific_args.arg_parser
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 :type _sphinx_paramlinks_pl_bolts.models.rl.dqn_model.DQN.forward.x: Tensor :param _sphinx_paramlinks_pl_bolts.models.rl.dqn_model.DQN.forward.x: environment state

Return type Tensor

Returns q values

static make_environment(*env_name*, *seed*)
Initialise gym environment

Parameters

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

Return type Env

Returns gym environment

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

Return type None

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 :rtype: 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 :type _sphinx_paramlinks_pl_bolts.models.rl.dqn_model.DQN.training_step.batch: Tuple[Tensor, Tensor] :param _sphinx_paramlinks_pl_bolts.models.rl.dqn_model.DQN.training_step.batch: current mini batch of replay data :param _sphinx_paramlinks_pl_bolts.models.rl.dqn_model.DQN.training_step._: batch number, not used

Return type OrderedDict

Returns Training loss and log metrics

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*, *learning_rate=0.0001*, *batch_size=32*, *replay_size=100000*, *warm_start_size=10000*, *avg_reward_len=100*, *min_episode_reward=-21*, *n_steps=1*, *seed=123*, *num_envs=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)
```

Parameters

- **env** (str) – gym environment tag
- **gpus** – number of gpus being used
- **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 espilon = eps_end
- **sync_rate** (int) – the number of iterations between syncing up the target network with the train network
- **gamma** (float) – discount factor
- **lr** – 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
- **sample_len** – the number of samples to pull from the dataset iterator and feed to the DataLoader

---

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

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)
```

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 espilon = 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
- **num_envs** (int) – number of environments to run the agent in at once

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*

build_networks()
Initializes the Dueling DQN train and target networks

Return type None

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*, *replay_size=100000*, *warm_start_size=10000*, *avg_reward_len=100*, *min_episode_reward=-21*, *n_steps=1*, *seed=123*, *num_envs=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)
```

Parameters

- **env** (*str*) – gym environment tag
- **gpus** – number of gpus being used
- **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 espilon = eps_end
- **sync_rate** (*int*) – the number of iterations between syncing up the target network with the train network
- **gamma** (*float*) – discount factor
- **lr** – 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
- **to fill the buffer with a starting point** (*training*) –
- **sample_len** – the number of samples to pull from the dataset iterator and feed to the DataLoader

---

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

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)
```

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 espilon = 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
- **num_envs** (int) – number of environments to run the agent in at once

---

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*

---

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

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*, *n_steps=1*, *seed=123*, *num_envs=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)

Args:
    env: gym environment tag
    gpus: number of gpus being used
    eps_start: starting value of epsilon for the epsilon-greedy exploration
    eps_end: final value of epsilon for the epsilon-greedy exploration
    eps_last_frame: the final frame in for the decrease of epsilon. At this frame␣
↪espilon = eps_end
    sync_rate: the number of iterations between syncing up the target network␣
↪with the train network
    gamma: discount factor
    learning_rate: learning rate
    batch_size: size of minibatch pulled from the DataLoader
    replay_size: total capacity of the replay buffer
    warm_start_size: how many random steps through the environment to be carried␣
↪out at the start of
        training to fill the buffer with a starting point
    num_samples: the number of samples to pull from the dataset iterator and feed␣
↪to the DataLoader

.. note::
```

(continues on next page)

(continued from previous page)

```
    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`
```

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)
```

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 espilon = 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
- **num_envs** (int) – number of environments to run the agent in at once

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*

_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 :rtype: 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.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*, *num_envs=1*, *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("PongNoFrameskip-v4")
```

Train:

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

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

---

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*

_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 cargs 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]

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.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*, *num_envs=4*, *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("PongNoFrameskip-v4")
```

Train:

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

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

---

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*

_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 cargs added

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*)

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** (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.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)
```

Subpackages

pl_bolts.models.self_supervised.amdim package

Submodules

pl_bolts.models.self_supervised.amdim.amdim_module module

class pl_bolts.models.self_supervised.amdim.amdim_module.**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*, ***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.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*, *encoder_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.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_ftr*, *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.ssl_datasets module

class pl_bolts.models.self_supervised.amdim.ssl_datasets.**CIFAR10Mixed**(*root*, *split='val'*, *transform=None*, *target_transform=None*, *download=False*, *nb_labeled_per_class=None*, *val_pct=0.1*)

Bases: *pl_bolts.models.self_supervised.amdim.ssl_datasets.SSLDatasetMixin*, torchvision.datasets.CIFAR10

class pl_bolts.models.self_supervised.amdim.ssl_datasets.**SSLDatasetMixin**

Bases: abc.ABC

classmethod deterministic_shuffle(*x*, *y*)

classmethod generate_train_val_split(*examples*, *labels*, *pct_val*)

Splits dataset uniformly across classes :param _sphinx_paramlinks_pl_bolts.models.self_supervised.amdim.ssl_datasets.SSL
:param _sphinx_paramlinks_pl_bolts.models.self_supervised.amdim.ssl_datasets.SSLDatasetMixin.generate_train_val_spli
:param _sphinx_paramlinks_pl_bolts.models.self_supervised.amdim.ssl_datasets.SSLDatasetMixin.generate_train_val_spli
:return:

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 :param _sphinx_paramlinks_pl_bolts.models.self_supervised.amdim.ssl_datasets.SSLDatasetMixin.select_nb_imgs_per_class.exar
:param _sphinx_paramlinks_pl_bolts.models.self_supervised.amdim.ssl_datasets.SSLDatasetMixin.select_nb_imgs_per_cla
:param _sphinx_paramlinks_pl_bolts.models.self_supervised.amdim.ssl_datasets.SSLDatasetMixin.select_nb_imgs_per_cla
:return:

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**(*heigh*
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-05*, *input_height=32*, *batch_size=32*, *num_workers=0*, *warmup_epochs=10*, *max_epochs=1000*, ***kwargs*)

Bases: pytorch_lightning.LightningModule

PyTorch Lightning implementation of Bring 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:

```
import pytorch_lightning as pl
from pl_bolts.models.self_supervised import BYOL
from pl_bolts.datamodules import CIFAR10DataModule
from pl_bolts.models.self_supervised.simclr.simclr_transforms import (
    SimCLREvalDataTransform, SimCLRTrainDataTransform)

# model
model = BYOL(num_classes=10)

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

trainer = pl.Trainer()
trainer.fit(model, 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(*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.models module

class pl_bolts.models.self_supervised.byol.models.**MLP**(*input_dim=2048*, *hidden_size=4096*, *output_dim=256*)

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

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**(*datamodule=None*, *encoder='cpc_encoder'*, *patch_size=8*, *patch_overlap=4*, *online_ft=True*, *task='cpc'*, *num_workers=4*, *learning_rate=0.0001*, *data_dir=''*, *batch_size=32*, *pretrained=None*, ***kwargs*)

Bases: pytorch_lightning.LightningModule

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

Example

```
>>> from pl_bolts.models.self_supervised import CPCV2
...
>>> model = CPCV2()
```

Train:

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

CLI command:

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

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

To Finetune:

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

Some uses:

```
# load resnet18 pretrained using CPC on imagenet
model = CPCV2(encoder='resnet18', pretrained=True)
resnet18 = model.encoder
renset18.freeze()

# it supportes any torchvision resnet
model = CPCV2(encoder='resnet50', pretrained=True)

# use it as a feature extractor
x = torch.rand(2, 3, 224, 224)
out = model(x)
```

Parameters

- **datamodule** (Optional[LightningDataModule]) – A Datamodule (optional). Otherwise set the dataloaders directly
- **encoder** (Union[str, Module, LightningModule]) – A string for any of the resnets in torchvision, or the original CPC encoder, or a custon nn.Module encoder
- **patch_size** (int) – How big to make the image patches
- **patch_overlap** (int) – How much overlap should each patch have.

- **online_ft** (int) – Enable a 1024-unit MLP to fine-tune online
- **task** (str) – Which self-supervised task to use ('cpc', 'amdim', etc...)
- **num_workers** (int) – num dataloader worksers
- **learning_rate** (int) – what learning rate to use
- **data_dir** (str) – where to store data
- **batch_size** (int) – batch size
- **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*)

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.**CPCResNet101**(*sample_batch*, *zero_init_residual=False*, *groups=1*, *width_per_group=64*, *replace_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*, *downsample_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*, *dilation=1*)

3x3 convolution with padding

pl_bolts.models.self_supervised.cpc.transforms module

class pl_bolts.models.self_supervised.cpc.transforms.**CPCEvalTransformsCIFAR10**(*patch_size=8*, *overlap=4*)

Bases: object

Transforms used for CPC:

Parameters

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

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())
```

__call__(*inp*)

Call self as a function.

class pl_bolts.models.self_supervised.cpc.transforms.**CPCEvalTransformsImageNet128**(*patch_size=*, *overlap=16*)

Bases: object

Transforms used for CPC:

Parameters

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

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())
```

__call__(*inp*)
Call self as a function.

class pl_bolts.models.self_supervised.cpc.transforms.**CPCEvalTransformsSTL10**(*patch_size=16*, *overlap=8*)

Bases: object

Transforms used for CPC:

Parameters

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

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())
```

__call__(*inp*)
Call self as a function.

class pl_bolts.models.self_supervised.cpc.transforms.**CPCTrainTransformsCIFAR10**(*patch_size=8*, *overlap=4*)

Bases: object

Transforms used for CPC:

Parameters

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

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())
```

__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:

Parameters

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

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())
```

__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:

Parameters

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

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())
```

__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

class pl_bolts.models.self_supervised.moco.moco2_module.**MocoV2**(*base_encoder='resnet18'*, *emb_dim=128*, *num_negatives=65536*, *encoder_momentum=0.999*, *softmax_temperature=0.07*, *learning_rate=0.03*, *momentum=0.9*, *weight_decay=0.0001*, *datamodule=None*, *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()
```

Train:

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

CLI command:

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

# imagenet
python moco2_module.py
    --gpus 8
    --dataset imagenet2012
```

(continues on next page)

(continued from previous page)

```
    --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** (Optional[LightningDataModule]) – 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: 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

training_step(*batch*, *batch_idx*)

validation_epoch_end(*outputs*)

validation_step(*batch*, *batch_idx*)

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.**DensenetEncoder**

Bases: torch.nn.Module

forward(*x*)

class pl_bolts.models.self_supervised.simclr.simclr_module.**Projection**(*input_dim=2048*, *hidden_dim=2048*, *output_dim=128*)

Bases: torch.nn.Module

forward(*x*)

class pl_bolts.models.self_supervised.simclr.simclr_module.**SimCLR**(*batch_size*, *num_samples*, *warmup_epochs=10*, *lr=0.0001*, *opt_weight_decay=1e-06*, *loss_temperature=0.5*, ***kwargs*)

Bases: pytorch_lightning.LightningModule

Parameters

- **batch_size** – the batch size
- **num_samples** – num samples in the dataset
- **warmup_epochs** – 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_encoder()

setup(*stage*)

shared_step(*batch*, *batch_idx*)

training_step(*batch*, *batch_idx*)

validation_step(*batch*, *batch_idx*)

pl_bolts.models.self_supervised.simclr.simclr_transforms module

class pl_bolts.models.self_supervised.simclr.simclr_transforms.**GaussianBlur**(*kernel_size*, *min=0.1*, *max=2.0*)

Bases: object

__call__(*sample*)
Call self as a function.

class pl_bolts.models.self_supervised.simclr.simclr_transforms.**SimCLREvalDataTransform**(*input_s=1*)

Bases: object

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)
```

__call__(*sample*)
Call self as a function.

class pl_bolts.models.self_supervised.simclr.simclr_transforms.**SimCLRTrainDataTransform**(*input_s=1*)

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)
```

(continues on next page)

(continued from previous page)

```
x = sample()
(xi, xj) = transform(x)
```

__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*, *replace_stride_with_dilation=None*, *norm_layer=None*, *return_all_feature_maps=False*)
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" :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet18.pretrained: If True, returns a model pretrained on ImageNet :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet18.pretrained: bool :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet18.progress: If True, displays a progress bar of the download to stderr :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet18.progress: bool

pl_bolts.models.self_supervised.resnets.**resnet34**(*pretrained=False*, *progress=True*, ***kwargs*)
ResNet-34 model from "Deep Residual Learning for Image Recognition" :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet34.pretrained: If True, returns a model pretrained on ImageNet :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet34.pretrained: bool :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet34.progress: If True, displays a progress bar of the download to stderr :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet34.progress: bool

pl_bolts.models.self_supervised.resnets.**resnet50**(*pretrained=False*, *progress=True*, ***kwargs*)

ResNet-50 model from "Deep Residual Learning for Image Recognition" :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet50.pretrained: If True, returns a model pre-trained on ImageNet :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet50.pretrained: bool :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet50.progress: If True, displays a progress bar of the download to stderr :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet50.progress: bool

pl_bolts.models.self_supervised.resnets.**resnet50_bn**(*pretrained=False*, *progress=True*, ***kwargs*)

ResNet-50 model from "Deep Residual Learning for Image Recognition" :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet50_bn.pretrained: If True, returns a model pre-trained on ImageNet :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet50_bn.pretrained: bool :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet50_bn.progress: If True, displays a progress bar of the download to stderr :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet50_bn.progress: bool

pl_bolts.models.self_supervised.resnets.**resnet101**(*pretrained=False*, *progress=True*, ***kwargs*)

ResNet-101 model from "Deep Residual Learning for Image Recognition" :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet101.pretrained: If True, returns a model pre-trained on ImageNet :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet101.pretrained: bool :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet101.progress: If True, displays a progress bar of the download to stderr :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet101.progress: bool

pl_bolts.models.self_supervised.resnets.**resnet152**(*pretrained=False*, *progress=True*, ***kwargs*)

ResNet-152 model from "Deep Residual Learning for Image Recognition" :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet152.pretrained: If True, returns a model pre-trained on ImageNet :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet152.pretrained: bool :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet152.progress: If True, displays a progress bar of the download to stderr :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnet152.progress: bool

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" :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnext50_32x4d.pretrained: If True, returns a model pre-trained on ImageNet :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnext50_32x4d.pretrained: bool :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnext50_32x4d.progress: If True, displays a progress bar of the download to stderr :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnext50_32x4d.progress: bool

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" :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnext101_32x8d.pretrained: If True, returns a model pre-trained on ImageNet :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnext101_32x8d.pretrained: bool :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnext101_32x8d.progress: If True, displays a progress bar of the download to stderr :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.resnext101_32x8d.progress: bool

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. :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.wide_resnet50_2.pretrained: If True, returns a model pre-trained on ImageNet :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.wide_resnet50_2.pretrained: bool :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.wide_resnet50_2.progress: If True, displays a progress bar of the download to stderr :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.wide_resnet50_2.progress: bool

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. :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.wide_resnet101_2.pretrained: If True, returns a model pre-trained on ImageNet :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.wide_resnet101_2.pretrained: bool :param _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.wide_resnet101_2.progress: If True, displays a progress bar of the download to stderr :type _sphinx_paramlinks_pl_bolts.models.self_supervised.resnets.wide_resnet101_2.progress: bool

pl_bolts.models.self_supervised.ssl_finetuner module

class pl_bolts.models.self_supervised.ssl_finetuner.**SSLFineTuner**(*backbone*, *in_features*, *num_classes*, *hidden_dim=1024*)

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)
```

(continues on next page)

(continued from previous page)

```
# train
trainer = pl.Trainer()
trainer.fit(finetuner, dm)

# test
trainer.test(datamodule=dm)
```

Parameters

- **backbone** – a pretrained model
- **in_features** – feature dim of backbone outputs
- **num_classes** – classes of the dataset
- **hidden_dim** – 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

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**(*datamodule=None*, *embed_dim=16*, *heads=2*, *layers=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.)
–embed_dim	16	512
–heads	2	8
–layers	8	24
–pixels	28	32
–vocab_size	16	512
–num_classes	10	10
–batch_size	64	128
–learning_rate	0.01	0.01
–steps	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

- **datamodule** (Optional[LightningDataModule]) – LightningDataModule
- **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

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*, *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*)

27.5.2 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_epoch_end(*outputs*)

test_step(*batch*, *batch_idx*)

train_dataloader()

training_step(*batch*, *batch_idx*)

val_dataloader()

validation_epoch_end(*outputs*)

validation_step(*batch*, *batch_idx*)

pl_bolts.models.mnist_module.**run_cli**()

27.6 pl_bolts.losses package

27.6.1 Submodules

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

forward(*anchor_representations*, *positive_representations*, *mask_mat*)
Compute the NCE scores for predicting r_src->r_trg. :param _sphinx_paramlinks_pl_bolts.losses.self_supervised_learning.AmdimNCELoss.forward.anchor_representations: (batch_size, emb_dim) :param _sphinx_paramlinks_pl_bolts.losses.self_supervised_learning.AmdimNCELoss.forward.posi (emb_dim, n_batch * w* h) (ie: nb_feat_vectors x embedding_dim) :param

_sphinx_paramlinks_pl_bolts.losses.self_supervised_learning.AmdimNCELoss.forward.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*, *embed_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*, *bidirectional=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))
```

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

```
# 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)
```

_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]

27.7 pl_bolts.loggers package

Collection of PyTorchLightning loggers

class pl_bolts.loggers.**TrainsLogger**(*project_name=None*, *task_name=None*, *task_type='training'*, *reuse_last_task_id=True*, *output_uri=None*, *auto_connect_arg_parser=True*, *auto_connect_frameworks=True*, *auto_resource_monitoring=True*)
Bases: pytorch_lightning.loggers.base.LightningLoggerBase

Log using allegro.ai TRAINS. Install it with pip:

```
pip install trains
```

Example

```
>>> from pytorch_lightning import Trainer
>>> trains_logger = TrainsLogger(
...     project_name='pytorch lightning',
...     task_name='default',
...     output_uri='.',
... )
TRAINS Task: ...
TRAINS results page: ...
>>> trainer = Trainer(logger=trains_logger)
```

Use the logger anywhere in your `LightningModule` as follows:

```
>>> from pytorch_lightning import LightningModule
>>> class LitModel(LightningModule):
...     def training_step(self, batch, batch_idx):
...         # example
...         self.logger.experiment.whatever_trains_supports(...)
...
...     def any_lightning_module_function_or_hook(self):
...         self.logger.experiment.whatever_trains_supports(...)
```

Parameters

- **project_name** (Optional[str]) – The name of the experiment's project. Defaults to None.
- **task_name** (Optional[str]) – The name of the experiment. Defaults to None.
- **task_type** (str) – The name of the experiment. Defaults to 'training'.
- **reuse_last_task_id** (bool) – Start with the previously used task id. Defaults to True.
- **output_uri** (Optional[str]) – Default location for output models. Defaults to None.
- **auto_connect_arg_parser** (bool) – Automatically grab the ArgumentParser and connect it with the task. Defaults to True.
- **auto_connect_frameworks** (bool) – If True, automatically patch to trains backend. Defaults to True.
- **auto_resource_monitoring** (bool) – If True, machine vitals will be sent along side the task scalars. Defaults to True.

Examples

```
>>> logger = TrainsLogger("pytorch lightning", "default", output_uri=".")
TRAINS Task: ...
TRAINS results page: ...
>>> logger.log_metrics({"val_loss": 1.23}, step=0)
>>> logger.log_text("sample test")
sample test
>>> import numpy as np
>>> logger.log_artifact("confusion matrix", np.ones((2, 3)))
>>> logger.log_image("passed", "Image 1", np.random.randint(0, 255, (200, 150, 3),
↪ dtype=np.uint8))
```

classmethod bypass_mode()
Returns the bypass mode state.

Note: *GITHUB_ACTIONS* env will automatically set bypass_mode to `True` unless overridden specifically with `TrainsLogger.set_bypass_mode(False)`.

Return type `bool`

Returns If True, all outside communication is skipped.

finalize(*status=None*)

Return type `None`

log_artifact(*name*, *artifact*, *metadata=None*, *delete_after_upload=False*)
Save an artifact (file/object) in TRAINS experiment storage.

Parameters

- **name** (`str`) – Artifact name. Notice! it will override the previous artifact if the name already exists.
- **artifact** (`Union[str, Path, Dict[str, Any], ndarray, Image]`) – Artifact object to upload. Currently supports:
  - string / `pathlib.Path` are treated as path to artifact file to upload If a wildcard or a folder is passed, a zip file containing the local files will be created and uploaded.
  - dict will be stored as .json file and uploaded
  - `pandas.DataFrame` will be stored as .csv.gz (compressed CSV file) and uploaded
  - `numpy.ndarray` will be stored as .npz and uploaded
  - `PIL.Image.Image` will be stored to .png file and uploaded
- **metadata** (`Optional[Dict[str, Any]]`) – Simple key/value dictionary to store on the artifact. Defaults to `None`.
- **delete_after_upload** (`bool`) – If `True`, the local artifact will be deleted (only applies if `artifact` is a local file). Defaults to `False`.

Return type `None`

log_hyperparams(*params*)
Log hyperparameters (numeric values) in TRAINS experiments.

Parameters **params** (Union[Dict[str, Any], Namespace]) – The hyperparameters that passed through the model.

Return type None

log_image(*title*, *series*, *image*, *step=None*)
Log Debug image in TRAINS experiment

Parameters

- **title** (str) – The title of the debug image, i.e. "failed", "passed".
- **series** (str) – The series name of the debug image, i.e. "Image 0", "Image 1".
- **image** (Union[str, ndarray, Image, Tensor]) – Debug image to log. If numpy.ndarray or torch.Tensor, the image is assumed to be the following:
  - shape: CHW
  - color space: RGB
  - value range: [0., 1.] (float) or [0, 255] (uint8)
- **step** (Optional[int]) – Step number at which the metrics should be recorded. Defaults to None.

Return type None

log_metric(*title*, *series*, *value*, *step=None*)
Log metrics (numeric values) in TRAINS experiments. This method will be called by the users.

Parameters

- **title** (str) – The title of the graph to log, e.g. loss, accuracy.
- **series** (str) – The series name in the graph, e.g. classification, localization.
- **value** (float) – The value to log.
- **step** (Optional[int]) – Step number at which the metrics should be recorded. Defaults to None.

Return type None

log_metrics(*metrics*, *step=None*)
Log metrics (numeric values) in TRAINS experiments. This method will be called by Trainer.

Parameters

- **metrics** (Dict[str, float]) – The dictionary of the metrics. If the key contains "/", it will be split by the delimiter, then the elements will be logged as "title" and "series" respectively.
- **step** (Optional[int]) – Step number at which the metrics should be recorded. Defaults to None.

Return type None

log_text(*text*)
Log console text data in TRAINS experiment.

Parameters **text** (str) – The value of the log (data-point).

Return type None

classmethod set_bypass_mode(*bypass*)
Will bypass all outside communication, and will drop all logs. Should only be used in "standalone mode", when there is no access to the *trains-server*.

Parameters **bypass** (bool) – If True, all outside communication is skipped.

Return type None

classmethod set_credentials(*api_host=None*, *web_host=None*, *files_host=None*, *key=None*, *secret=None*)
Set new default TRAINS-server host and credentials. These configurations could be overridden by either OS environment variables or trains.conf configuration file.

Note: Credentials need to be set *prior* to Logger initialization.

Parameters

- **api_host** (Optional[str]) – Trains API server url, example: host='http://localhost:8008'
- **web_host** (Optional[str]) – Trains WEB server url, example: host='http://localhost:8080'
- **files_host** (Optional[str]) – Trains Files server url, example: host='http://localhost:8081'
- **key** (Optional[str]) – user key/secret pair, example: key='thisisakey123'
- **secret** (Optional[str]) – user key/secret pair, example: secret='thisisseceret123'

Return type None

_bypass = None

property experiment
Actual TRAINS object. To use TRAINS features in your LightningModule do the following.

Example:

```
self.logger.experiment.some_trains_function()
```

Return type Task

property id
ID is a uuid (string) representing this specific experiment in the entire system.

Return type Optional[str]

property name
Name is a human readable non-unique name (str) of the experiment.

Return type Optional[str]

property version

Return type Optional[str]

27.7.1 Submodules

pl_bolts.loggers.trains module

TRAINS

class pl_bolts.loggers.trains.**TrainsLogger**(*project_name=None*, *task_name=None*, *task_type='training'*, *reuse_last_task_id=True*, *output_uri=None*, *auto_connect_arg_parser=True*, *auto_connect_frameworks=True*, *auto_resource_monitoring=True*)

Bases: pytorch_lightning.loggers.base.LightningLoggerBase

Log using allegro.ai TRAINS. Install it with pip:

```
pip install trains
```

Example

```
>>> from pytorch_lightning import Trainer
>>> trains_logger = TrainsLogger(
...     project_name='pytorch lightning',
...     task_name='default',
...     output_uri='.',
... )
TRAINS Task: ...
TRAINS results page: ...
>>> trainer = Trainer(logger=trains_logger)
```

Use the logger anywhere in your LightningModule as follows:

```
>>> from pytorch_lightning import LightningModule
>>> class LitModel(LightningModule):
...     def training_step(self, batch, batch_idx):
...         # example
...         self.logger.experiment.whatever_trains_supports(...)
...
...     def any_lightning_module_function_or_hook(self):
...         self.logger.experiment.whatever_trains_supports(...)
```

Parameters

- **project_name** (Optional[str]) – The name of the experiment's project. Defaults to None.
- **task_name** (Optional[str]) – The name of the experiment. Defaults to None.
- **task_type** (str) – The name of the experiment. Defaults to 'training'.
- **reuse_last_task_id** (bool) – Start with the previously used task id. Defaults to True.
- **output_uri** (Optional[str]) – Default location for output models. Defaults to None.
- **auto_connect_arg_parser** (bool) – Automatically grab the ArgumentParser and connect it with the task. Defaults to True.

- **auto_connect_frameworks** (bool) – If True, automatically patch to trains backend. Defaults to True.
- **auto_resource_monitoring** (bool) – If True, machine vitals will be sent along side the task scalars. Defaults to True.

Examples

```
>>> logger = TrainsLogger("pytorch lightning", "default", output_uri=".")
TRAINS Task: ...
TRAINS results page: ...
>>> logger.log_metrics({"val_loss": 1.23}, step=0)
>>> logger.log_text("sample test")
sample test
>>> import numpy as np
>>> logger.log_artifact("confusion matrix", np.ones((2, 3)))
>>> logger.log_image("passed", "Image 1", np.random.randint(0, 255, (200, 150, 3),
↪ dtype=np.uint8))
```

classmethod bypass_mode()
Returns the bypass mode state.

Note: *GITHUB_ACTIONS* env will automatically set bypass_mode to True unless overridden specifically with TrainsLogger.set_bypass_mode(False).

Return type bool

Returns If True, all outside communication is skipped.

finalize(*status=None*)

Return type None

log_artifact(*name*, *artifact*, *metadata=None*, *delete_after_upload=False*)
Save an artifact (file/object) in TRAINS experiment storage.

Parameters

- **name** (str) – Artifact name. Notice! it will override the previous artifact if the name already exists.
- **artifact** (Union[str, Path, Dict[str, Any], ndarray, Image]) – Artifact object to upload. Currently supports:
  - string / pathlib.Path are treated as path to artifact file to upload If a wildcard or a folder is passed, a zip file containing the local files will be created and uploaded.
  - dict will be stored as .json file and uploaded
  - pandas.DataFrame will be stored as .csv.gz (compressed CSV file) and uploaded
  - numpy.ndarray will be stored as .npz and uploaded
  - PIL.Image.Image will be stored to .png file and uploaded
- **metadata** (Optional[Dict[str, Any]]) – Simple key/value dictionary to store on the artifact. Defaults to None.

- **delete_after_upload** (bool) – If True, the local artifact will be deleted (only applies if artifact is a local file). Defaults to False.

Return type None

log_hyperparams(*params*)
Log hyperparameters (numeric values) in TRAINS experiments.

Parameters **params** (Union[Dict[str, Any], Namespace]) – The hyperparameters that passed through the model.

Return type None

log_image(*title*, *series*, *image*, *step=None*)
Log Debug image in TRAINS experiment

Parameters

- **title** (str) – The title of the debug image, i.e. "failed", "passed".
- **series** (str) – The series name of the debug image, i.e. "Image 0", "Image 1".
- **image** (Union[str, ndarray, Image, Tensor]) – Debug image to log. If numpy.ndarray or torch.Tensor, the image is assumed to be the following:
  - shape: CHW
  - color space: RGB
  - value range: [0., 1.] (float) or [0, 255] (uint8)
- **step** (Optional[int]) – Step number at which the metrics should be recorded. Defaults to None.

Return type None

log_metric(*title*, *series*, *value*, *step=None*)
Log metrics (numeric values) in TRAINS experiments. This method will be called by the users.

Parameters

- **title** (str) – The title of the graph to log, e.g. loss, accuracy.
- **series** (str) – The series name in the graph, e.g. classification, localization.
- **value** (float) – The value to log.
- **step** (Optional[int]) – Step number at which the metrics should be recorded. Defaults to None.

Return type None

log_metrics(*metrics*, *step=None*)
Log metrics (numeric values) in TRAINS experiments. This method will be called by Trainer.

Parameters

- **metrics** (Dict[str, float]) – The dictionary of the metrics. If the key contains "/", it will be split by the delimiter, then the elements will be logged as "title" and "series" respectively.
- **step** (Optional[int]) – Step number at which the metrics should be recorded. Defaults to None.

Return type None

log_text(*text*)
Log console text data in TRAINS experiment.

Parameters **text** (str) – The value of the log (data-point).

Return type None

classmethod set_bypass_mode(*bypass*)
Will bypass all outside communication, and will drop all logs. Should only be used in "standalone mode", when there is no access to the *trains-server*.

Parameters **bypass** (bool) – If True, all outside communication is skipped.

Return type None

classmethod set_credentials(*api_host=None*, *web_host=None*, *files_host=None*, *key=None*, *secret=None*)
Set new default TRAINS-server host and credentials. These configurations could be overridden by either OS environment variables or trains.conf configuration file.

Note: Credentials need to be set *prior* to Logger initialization.

Parameters

- **api_host** (Optional[str]) – Trains API server url, example: host='http://localhost:8008'
- **web_host** (Optional[str]) – Trains WEB server url, example: host='http://localhost:8080'
- **files_host** (Optional[str]) – Trains Files server url, example: host='http://localhost:8081'
- **key** (Optional[str]) – user key/secret pair, example: key='thisisakey123'
- **secret** (Optional[str]) – user key/secret pair, example: secret='thisisseceret123'

Return type None

_bypass = None

property experiment
Actual TRAINS object. To use TRAINS features in your LightningModule do the following.

Example:

```
self.logger.experiment.some_trains_function()
```

Return type Task

property id
ID is a uuid (string) representing this specific experiment in the entire system.

Return type Optional[str]

property name
Name is a human readable non-unique name (str) of the experiment.

Return type Optional[str]

property version

Return type Optional[str]

27.8 pl_bolts.optimizers package

27.8.1 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()

update_p(*p*, *group*, *weight_decay*)

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.

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.

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(...)
```

_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]

27.9 pl_bolts.transforms package

27.9.1 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*, *overlap_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 [-max_translation, max_translation]. Fill the uncovered blank area with reflect padding.

__call__(*old_image*)
Call self as a function.

27.9.2 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**()

27.10 pl_bolts.utils package

27.10.1 Submodules

pl_bolts.utils.pretrained_weights module

pl_bolts.utils.pretrained_weights.**load_pretrained**(*model*, *class_name=None*)

pl_bolts.utils.self_supervised module

pl_bolts.utils.self_supervised.**torchvision_ssl_encoder**(*name*, *pretrained=False*, *return_all_feature_maps=False*)

pl_bolts.utils.semi_supervised module

class pl_bolts.utils.semi_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()
```

forward(*x*)

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** (ndarray) – input features
- **Y** (list) – mixed labels (ints)
- **batch_size** (int) – the ultimate batch size

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

pl_bolts.utils.shaping module

pl_bolts.utils.shaping.**tile**(*a*, *dim*, *n_tile*)

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

G

M

N

O

P

R

S

T

U

V

W