Lightning-Bolts Documentation

Release 0.7.0

Lightning AI et al.

Jun 30, 2023

START HERE

1 Installation 1

2 Introduction Guide 3

3 Monitoring Callbacks 13

4 Torch ORT Callback 17

5 SparseML Callback 19

6 Self-supervised Callbacks 21

7 Variational Callbacks 25

8 Vision Callbacks 27

9 Sklearn Datamodule 31

10 Vision DataModules 37

11 Debug Datasets 59

12 AsynchronousLoader 63

13 Reinforcement Learning 65

14 How to use models 67

15 Autoencoders 75

16 Convolutional Architectures 93

17 GANs 115

18 Object Detection 149

19 Reinforcement Learning 169

20 Self-supervised Learning 203

21 Classic ML Models 265

22 Linear Warmup Cosine Annealing 279

23 Self-supervised learning 281

24 Semi-supervised learning 305

25 Self-supervised Learning 307

26 Contributing 311

27 PL Bolts Governance | Persons of interest 315

28 Bolts stability 317

29 Changelog 319

30 Indices and tables 331

Index 333

CHAPTER

ONE

INSTALLATION

You can install using pip

```
pip install lightning-bolts
```

Install bleeding-edge (no guarantees)

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

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

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

CHAPTER

TWO

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!

Note: Currently, Bolts is going through a major revision. For more information about it, see these GitHub issues (#819 and #839) and stability section

---

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
from pl_bolts.models.vision import GPT2, ImageGPT, PixelCNN
from pl_bolts.models.self_supervised import AMDIM, CPC_v2, SimCLR, Moco_v2
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.vision import ImageGPT
from pl_bolts.models.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)
```

(continues on next page)

(continued from previous page)

```
        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}
```

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

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

And train on any hardware accelerator

```
import pytorch_lightning as pl

model = ImageGPT(datamodule=FashionMNISTDataModule(PATH))

# cpus
pl.Trainer.fit(model)

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

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

Or pass in any dataset of your choice

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

2.1 Community Built

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

1. Rigorously tested (CPUs, GPUs, TPUs).
2. Rigorously documented.
3. Standardized via PyTorch Lightning.

4. Optimized for speed.
5. Checked for correctness.

2.1.1 How to contribute

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

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

To contribute:

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

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

2.1.2 Contribution ideas

Don't have something to contribute? Ping us on Slack or look at our Github issues!

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

---

2.2 When to use Bolts

2.2.1 For pretrained models

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

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

```
from pl_bolts.models.autoencoders import VAE
from pl_bolts.models.self_supervised import CPC_v2

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

# bolts are pretrained on different datasets
model2 = CPC_v2(encoder='resnet18', pretrained='imagenet128').freeze()
model3 = CPC_v2(encoder='resnet18', pretrained='stl10').freeze()
```

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

# which is better?
```

2.2.2 To finetune on your data

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

Example 1: Unfrozen finetune

```
# unfrozen finetune
model = CPC_v2(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 = CPC_v2(encoder='resnet18', pretrained='imagenet128')
resnet18 = model.encoder
resnet18.eval()

classifier = LogisticRegression(...)

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

    # UNFREEZE after 10 epochs
```

(continues on next page)

(continued from previous page)

```
    if epoch == 10:
        resnet18.unfreeze()
```

2.2.3 For research

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

```
from pl_bolts.models.vision import ImageGPT
from pl_bolts.models.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)
```

(continues on next page)

(continued from previous page)

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

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

        return loss
```

2.3 Callbacks

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

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

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

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

2.4 DataModules

In PyTorch, working with data has these major elements.

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

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

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

```
from pl_bolts.datamodules import ImagenetDataModule

dm = ImagenetDataModule(data_dir=PATH)

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

(continues on next page)

(continued from previous page)

```
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_diabetes
from pl_bolts.datamodules import SklearnDataModule

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

model = LitModel(datamodule)
```

2.5 Regression Heroes

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

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

```
from pl_bolts.models.regression import LogisticRegression

imagenet = ImagenetDataModule(PATH)

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

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

2.5.1 Linear Regression

Here's an example for Linear regression

```
import pytorch_lightning as pl
from pl_bolts.datamodules import SklearnDataModule
from sklearn.datasets import load_diabetes

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

# training runs training batches while validating against a validation set
model = LinearRegression()
trainer = pl.Trainer(num_gpus=8)
trainer.fit(model, train_dataloaders=loaders.train_dataloader(), val_dataloaders=loaders.
↪val_dataloader())
```

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

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

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

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

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

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

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

2.5.2 Logistic Regression

Here's an example for logistic regression

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

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

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

# fit
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

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

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

2.6 Regular PyTorch

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

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

2.7 Command line support

Any bolt module can also be trained from the command line

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

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

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

CHAPTER

THREE

MONITORING CALLBACKS

These callbacks give all sorts of useful information during training.

3.1 Print Table Metrics

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

class pl_bolts.callbacks.printing.**PrintTableMetricsCallback**
Bases: pytorch_lightning.callbacks.callback.Callback

> **Warning:** The feature PrintTableMetricsCallback is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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_train_epoch_end(*trainer*, *pl_module*)
Called when the train epoch ends.

To access all batch outputs at the end of the epoch, either:

1. Implement *training_epoch_end* in the *LightningModule* and access outputs via the module OR
2. Cache data across train batch hooks inside the callback implementation to post-process in this hook.

Return type None

3.2 Data Monitoring in LightningModule

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

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

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

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

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

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

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

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

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

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

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

3.3 Model Verification

3.3.1 Gradient-Check for Batch-Optimization

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

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

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

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

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

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

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

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

```
from pl_bolts.utils import BatchGradientVerification

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

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

CHAPTER

FOUR

TORCH ORT CALLBACK

Torch ORT converts your model into an optimized ONNX graph, speeding up training & inference when using NVIDIA or AMD GPUs. See installation instructions here.

This is primarily useful for when training with a Transformer model. The ORT callback works when a single model is specified as *self.model* within the `LightningModule` as shown below.

Note: Not all Transformer models are supported. See this table for supported models + branches containing fixes for certain models.

```
from pytorch_lightning import LightningModule, Trainer
from transformers import AutoModel

from pl_bolts.callbacks import ORTCallback


class MyTransformerModel(LightningModule):

    def __init__(self):
        super().__init__()
        self.model = AutoModel.from_pretrained('bert-base-cased')

    ...


model = MyTransformerModel()
trainer = Trainer(gpus=1, callbacks=ORTCallback())
trainer.fit(model)
```

For even easier setup and integration, have a look at our Lightning Flash integration for Text Classification, Translation and Summarization.

CHAPTER

FIVE

SPARSEML CALLBACK

SparseML allows you to leverage sparsity to improve inference times substantially.

SparseML requires you to fine-tune your model with the `SparseMLCallback` + a SparseML Recipe. By training with the `SparseMLCallback`, you can leverage the DeepSparse engine to exploit the introduced sparsity, resulting in large performance improvements.

> **Warning:** The SparseML callback requires the model to be ONNX exportable. This can be tricky when the model requires dynamic sequence lengths such as RNNs.

To use leverage SparseML & DeepSparse follow the below steps:

5.1 1. Choose your Sparse Recipe

To choose a recipe, have a look at recipes and Sparse Zoo.

It may be easier to infer a recipe via the UI dashboard using Sparsify which allows you to tweak and configure a recipe. This requires to import an ONNX model, which you can get from your `LightningModule` by doing `model.to_onnx(output_path)`.

5.2 2. Train with SparseMLCallback

```
from pytorch_lightning import LightningModule, Trainer
from pl_bolts.callbacks import SparseMLCallback

class MyModel(LightningModule):
    ...

model = MyModel()

trainer = Trainer(
    callbacks=SparseMLCallback(recipe_path='recipe.yaml')
)
```

5.3 3. Export to ONNX!

Using the helper function, we handle any quantization/pruning internally and export the model into ONNX format. Note this assumes either you have implemented the property `example_input_array` in the model or you must provide a sample batch as below.

```
import torch

model = MyModel()
...

# export the onnx model, using the `model.example_input_array`
SparseMLCallback.export_to_sparse_onnx(model, 'onnx_export/')

# export the onnx model, providing a sample batch
SparseMLCallback.export_to_sparse_onnx(model, 'onnx_export/', sample_batch=torch.randn(1,
↪ 128, 128, dtype=torch.float32))
```

Once your model has been exported, you can import this into either Sparsify or DeepSparse.

CHAPTER

SIX

SELF-SUPERVISED CALLBACKS

Useful callbacks for self-supervised learning models.

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

6.1 BYOLMAWeightUpdate

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

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

Bases: pytorch_lightning.callbacks.callback.Callback

Weight update rule from Bootstrap Your Own Latent (BYOL).

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

The PyTorch Lightning module being trained should have:

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

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

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

Example:

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

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

on_train_batch_end(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*)
Called when the train batch ends.

Note: The value `outputs["loss"]` here will be the normalized value w.r.t `accumulate_grad_batches` of the loss returned from `training_step`.

Return type None

update_tau(*pl_module*, *trainer*)
Update tau value for next update.

Return type None

update_weights(*online_net*, *target_net*)
Update target network parameters.

Return type None

6.2 SSLOnlineEvaluator

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

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

Warning: The feature SSLOnlineEvaluator is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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

Example:

```
# your datamodule must have 2 attributes
dm = DataModule()
dm.num_classes = ... # the num of classes in the datamodule
dm.name = ... # name of the datamodule (e.g. ImageNet, STL10, CIFAR10)

# your model must have 1 attribute
model = Model()
model.z_dim = ... # the representation dim

online_eval = SSLOnlineEvaluator(
    z_dim=model.z_dim
)
```

Parameters

- **z_dim** (int) – Representation dimension

- **drop_p** (float) – Dropout probability
- **hidden_dim** (Optional[int]) – Hidden dimension for the fine-tune MLP

load_state_dict(*state_dict*)
Called when loading a checkpoint, implement to reload callback state given callback's state_dict.

Parameters **state_dict** (Dict[str, Any]) – the callback state returned by state_dict.

Return type None

on_fit_start(*trainer*, *pl_module*)
Called when fit begins.

Return type None

on_train_batch_end(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*)
Called when the train batch ends.

Note: The value outputs["loss"] here will be the normalized value w.r.t accumulate_grad_batches of the loss returned from training_step.

Return type None

on_validation_batch_end(*trainer*, *pl_module*, *outputs*, *batch*, *batch_idx*, *dataloader_idx*)
Called when the validation batch ends.

Return type None

setup(*trainer*, *pl_module*, *stage=None*)
Called when fit, validate, test, predict, or tune begins.

Return type None

state_dict()
Called when saving a checkpoint, implement to generate callback's state_dict.

Return type dict

Returns A dictionary containing callback state.

CHAPTER

SEVEN

VARIATIONAL CALLBACKS

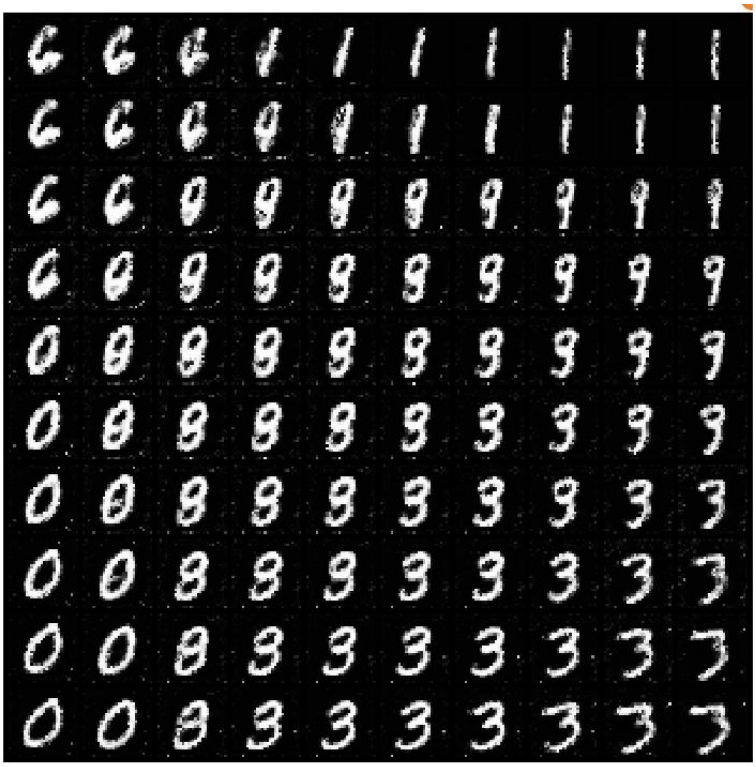
Useful callbacks for GANs, variational-autoencoders or anything with latent spaces.

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

7.1 Latent Dim Interpolator

Interpolates latent dims.

Example output:



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

Bases: pytorch_lightning.callbacks.callback.Callback

> **Warning:** The feature LatentDimInterpolator is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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

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

Example:

```
from pl_bolts.callbacks import LatentDimInterpolator

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

Parameters

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

on_train_epoch_end(*trainer*, *pl_module*)

Called when the train epoch ends.

To access all batch outputs at the end of the epoch, either:

1. Implement *training_epoch_end* in the *LightningModule* and access outputs via the module OR
2. Cache data across train batch hooks inside the callback implementation to post-process in this hook.

Return type None

CHAPTER

EIGHT

VISION CALLBACKS

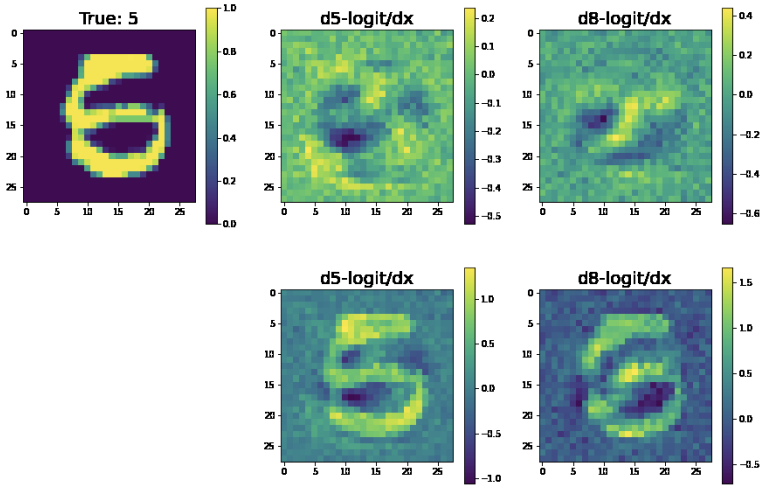
Useful callbacks for vision models.

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

8.1 Confused Logit

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

Example outputs:



class pl_bolts.callbacks.vision.confused_logit.**ConfusedLogitCallback**(*top_k*, *min_logit_value=5.0*, *logging_batch_interval=20*, *max_logit_difference=0.1*)

Bases: pytorch_lightning.callbacks.callback.Callback

Warning: The feature ConfusedLogitCallback is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may

change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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

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

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

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

Example:

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

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

Note: This callback supports tensorboard only right now.

Authored by:

- Alfredo Canziani

Parameters

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

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

Called when the train batch ends.

Note: The value `outputs["loss"]` here will be the normalized value w.r.t `accumulate_grad_batches` of the loss returned from `training_step`.

Return type None

8.2 Tensorboard Image Generator

Generates images from a generative model and plots to tensorboard

class pl_bolts.callbacks.vision.image_generation.**TensorboardGenerativeModelImageSampler**(*num_samples=3*, *nrow=8*, *padding=2*, *normalize=False*, *norm_range=None*, *scale_each=False*, *pad_value=0*)

Bases: pytorch_lightning.callbacks.callback.Callback

> **Warning:** The feature TensorboardGenerativeModelImageSampler is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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

Requirements:

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

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

Example:

```
from pl_bolts.callbacks import TensorboardGenerativeModelImageSampler

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

Parameters

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

- **pad_value** (int) – Value for the padded pixels. Default: 0.

on_train_epoch_end(*trainer*, *pl_module*)
Called when the train epoch ends.

To access all batch outputs at the end of the epoch, either:

1. Implement *training_epoch_end* in the *LightningModule* and access outputs via the module OR
2. Cache data across train batch hooks inside the callback implementation to post-process in this hook.

Return type None

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_diabetes
from pl_bolts.datamodules import SklearnDataModule

X, y = load_diabetes(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_diabetes
from pl_bolts.datamodules import SklearnDataset

X, y = load_diabetes(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: Generic[torch.utils.data.dataset.T_co]

> **Warning:** The feature SklearnDataset is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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_diabetes
>>> from pl_bolts.datamodules import SklearnDataset
...
>>> X, y = load_diabetes(return_X_y=True)
>>> dataset = SklearnDataset(X, y)
>>> len(dataset)
442
```

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=0*, *random_state=1234*, *shuffle=True*, *batch_size=16*, *pin_memory=True*, *drop_last=False*, **args*, ***kwargs*)

Bases: pytorch_lightning.core.datamodule.LightningDataModule

Warning: The feature SklearnDataModule is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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_diabetes
>>> from pl_bolts.datamodules import SklearnDataModule
...
>>> X, y = load_diabetes(return_X_y=True)
>>> loaders = SklearnDataModule(X, y, batch_size=32)
...
>>> # train set
>>> train_loader = loaders.train_dataloader()
>>> len(train_loader.dataset)
310
>>> len(train_loader)
10
>>> # validation set
>>> val_loader = loaders.val_dataloader()
>>> len(val_loader.dataset)
88
>>> len(val_loader)
3
>>> # test set
>>> test_loader = loaders.test_dataloader()
>>> len(test_loader.dataset)
44
>>> len(test_loader)
2
```

prepare_data_per_node
If True, each LOCAL_RANK=0 will call prepare data. Otherwise only NODE_RANK=0, LOCAL_RANK=0 will prepare data.

allow_zero_length_dataloader_with_multiple_devices
If True, dataloader with zero length within local rank is allowed. Default value is False.

test_dataloader()
Implement one or multiple PyTorch DataLoaders for testing.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

> **Warning:** do not assign state in prepare_data

- `test()`
- `prepare_data()`
- `setup()`

Note: Lightning adds the correct sampler for distributed and arbitrary hardware. There is no need to set

it yourself.

Return type DataLoader

Returns A torch.utils.data.DataLoader or a sequence of them specifying testing samples.

Example:

```
def test_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=False, transform=transform,
                    download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=False
    )

    return loader

# can also return multiple dataloaders
def test_dataloader(self):
    return [loader_a, loader_b, ..., loader_n]
```

Note: If you don't need a test dataset and a `test_step()`, you don't need to implement this method.

Note: In the case where you return multiple test dataloaders, the `test_step()` will have an argument `dataloader_idx` which matches the order here.

`train_dataloader()`

Implement one or more PyTorch DataLoaders for training.

Return type DataLoader

Returns A collection of torch.utils.data.DataLoader specifying training samples. In the case of multiple dataloaders, please see this section.

The dataloader you return will not be reloaded unless you set `reload_dataloaders_every_n_epochs` to a positive integer.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `fit()`

- prepare_data()
- setup()

Note: Lightning adds the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

Example:

```
# single dataloader
def train_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=True, transform=transform,
                    download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=True
    )
    return loader

# multiple dataloaders, return as list
def train_dataloader(self):
    mnist = MNIST(...)
    cifar = CIFAR(...)
    mnist_loader = torch.utils.data.DataLoader(
        dataset=mnist, batch_size=self.batch_size, shuffle=True
    )
    cifar_loader = torch.utils.data.DataLoader(
        dataset=cifar, batch_size=self.batch_size, shuffle=True
    )
    # each batch will be a list of tensors: [batch_mnist, batch_cifar]
    return [mnist_loader, cifar_loader]

# multiple dataloader, return as dict
def train_dataloader(self):
    mnist = MNIST(...)
    cifar = CIFAR(...)
    mnist_loader = torch.utils.data.DataLoader(
        dataset=mnist, batch_size=self.batch_size, shuffle=True
    )
    cifar_loader = torch.utils.data.DataLoader(
        dataset=cifar, batch_size=self.batch_size, shuffle=True
    )
    # each batch will be a dict of tensors: {'mnist': batch_mnist, 'cifar': batch_
↪cifar}
    return {'mnist': mnist_loader, 'cifar': cifar_loader}
```

val_dataloader()

Implement one or multiple PyTorch DataLoaders for validation.

The dataloader you return will not be reloaded unless you set `reload_dataloaders_every_n_epochs` to a positive integer.

It's recommended that all data downloads and preparation happen in `prepare_data()`.

- `fit()`
- `validate()`
- `prepare_data()`
- `setup()`

Note: Lightning adds the correct sampler for distributed and arbitrary hardware There is no need to set it yourself.

Return type `DataLoader`

Returns A `torch.utils.data.DataLoader` or a sequence of them specifying validation samples.

Examples:

```
def val_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=False,
                    transform=transform, download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=False
    )

    return loader

# can also return multiple dataloaders
def val_dataloader(self):
    return [loader_a, loader_b, ..., loader_n]
```

Note: If you don't need a validation dataset and a `validation_step()`, you don't need to implement this method.

Note: In the case where you return multiple validation dataloaders, the `validation_step()` will have an argument `dataloader_idx` which matches the order here.

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 BinaryEMNIST

class pl_bolts.datamodules.binary_emnist_datamodule.**BinaryEMNISTDataModule**(*data_dir=None*, *split='mnist'*, *val_split=0.2*, *num_workers=0*, *normalize=False*, *batch_size=32*, *seed=42*, *shuffle=True*, *pin_memory=True*, *drop_last=False*, *strict_val_split=False*, **args*, ***kwargs*)

Bases: *pl_bolts.datamodules.emnist_datamodule.EMNISTDataModule*

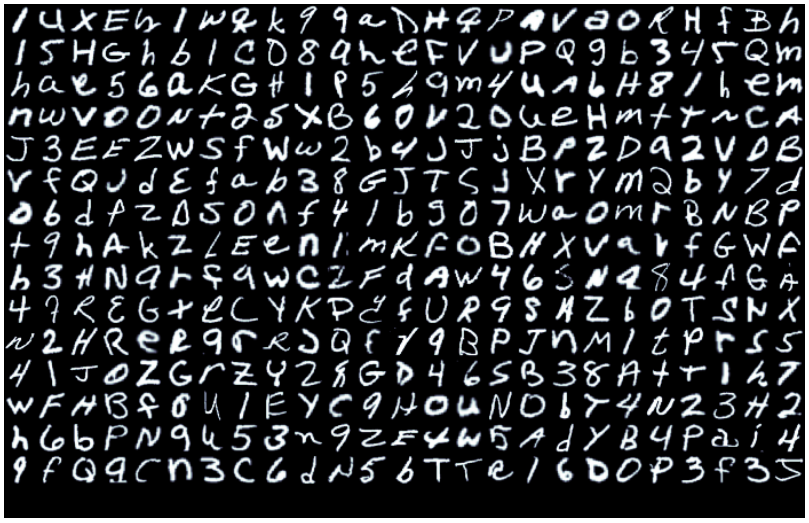
Please see *EMNISTDataModule* for more details.

Example:

```
from pl_bolts.datamodules import BinaryEMNISTDataModule
dm = BinaryEMNISTDataModule('.')
model = LitModel()
Trainer().fit(model, datamodule=dm)
```

Parameters

- **data_dir** (Optional[str]) – Where to save/load the data.
- **split** (str) – The dataset has 6 different splits: `byclass`, `bymerge`, `balanced`, `letters`, `digits` and `mnist`. This argument is passed to `torchvision.datasets.EMNIST`.



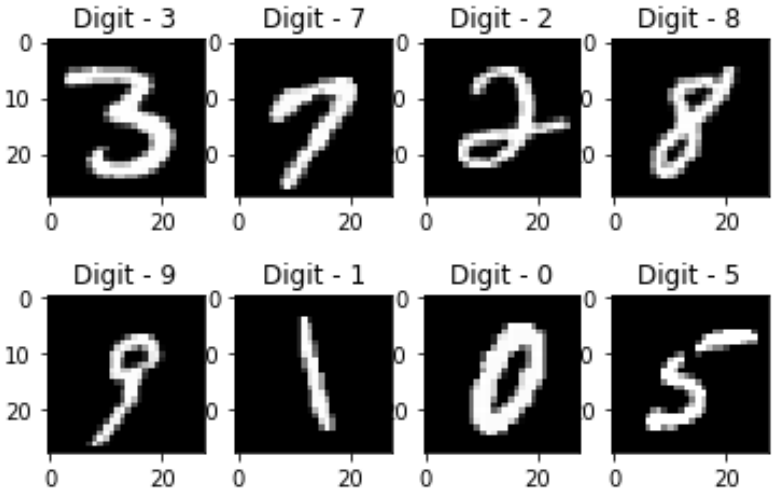
- **val_split** (Union[int, float]) – Percent (float) or number (int) of samples to use for the validation split.
- **num_workers** (int) – How many workers to use for loading data
- **normalize** (bool) – If True, applies image normalize.
- **batch_size** (int) – How many samples per batch to load.
- **seed** (int) – Random seed to be used for train/val/test splits.
- **shuffle** (bool) – If True, shuffles the train data every epoch.
- **pin_memory** (bool) – If True, the data loader will copy Tensors into CUDA pinned memory before returning them.
- **drop_last** (bool) – If True, drops the last incomplete batch.
- **strict_val_split** (bool) – If True, uses the validation split defined in the paper and ignores val_split. Note that it only works with "balanced", "digits", "letters", "mnist" splits.

dataset_cls
alias of pl_bolts.datasets.emnist_dataset.BinaryEMNIST

10.1.2 BinaryMNIST

class pl_bolts.datamodules.binary_mnist_datamodule.**BinaryMNISTDataModule**(*data_dir=None*, *val_split=0.2*, *num_workers=0*, *normalize=False*, *batch_size=32*, *seed=42*, *shuffle=True*, *pin_memory=True*, *drop_last=False*, **args*, ***kwargs*)

Bases: pl_bolts.datamodules.vision_datamodule.VisionDataModule



Specs:

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

Binary MNIST, train, val, test splits and transforms

Transforms:

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

Example:

```
from pl_bolts.datamodules import BinaryMNISTDataModule

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

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

Parameters

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

- **drop_last** (bool) – If true drops the last incomplete batch

dataset_cls
alias of pl_bolts.datasets.mnist_dataset.BinaryMNIST

default_transforms()
Default transform for the dataset.

Return type Callable

property num_classes: int
Return: 10

Return type int

10.1.3 CityScapes

class pl_bolts.datamodules.cityscapes_datamodule.**CityscapesDataModule**(*data_dir*, *quality_mode='fine'*, *target_type='instance'*, *num_workers=0*, *batch_size=32*, *seed=42*, *shuffle=True*, *pin_memory=True*, *drop_last=False*, *train_transforms=None*, *val_transforms=None*, *test_transforms=None*, *target_transforms=None*, **args*, ***kwargs*)

Bases: pytorch_lightning.core.datamodule.LightningDataModule

Warning: The feature CityscapesDataModule is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html



Standard Cityscapes, train, val, test splits and transforms

Note: You need to have downloaded the Cityscapes dataset first and provide the path to where it is saved.
You can download the dataset here: https://www.cityscapes-dataset.com/

Specs:

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

Transforms:

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

Example:

```
from pl_bolts.datamodules import CityscapesDataModule

dm = CityscapesDataModule(PATH)
model = LitModel()

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

Or you can set your own transforms

Example:

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

Parameters

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

test_dataloader()

Cityscapes test set.

Return type DataLoader

train_dataloader()
Cityscapes train set.

Return type DataLoader

val_dataloader()
Cityscapes val set.

Return type DataLoader

property num_classes: int
Return: 30

Return type int

10.1.4 CIFAR-10

class pl_bolts.datamodules.cifar10_datamodule.**CIFAR10DataModule**(*data_dir=None*, *val_split=0.2*, *num_workers=0*, *normalize=False*, *batch_size=32*, *seed=42*, *shuffle=True*, *pin_memory=True*, *drop_last=False*, **args*, ***kwargs*)

Bases: pl_bolts.datamodules.vision_datamodule.VisionDataModule

Specs:

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

Standard CIFAR10, train, val, test splits and transforms

Transforms:

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

Example:

```
from pl_bolts.datamodules import CIFAR10DataModule

dm = CIFAR10DataModule(PATH)
model = LitModel()

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

Or you can set your own transforms

Example:

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

Parameters

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

dataset_cls
alias of torchvision.datasets.cifar.CIFAR10

default_transforms()
Default transform for the dataset.

Return type Callable

property num_classes: int
Return: 10

Return type int

10.1.5 EMNIST

class pl_bolts.datamodules.emnist_datamodule.**EMNISTDataModule**(*data_dir=None*, *split='mnist'*, *val_split=0.2*, *num_workers=0*, *normalize=False*, *batch_size=32*, *seed=42*, *shuffle=True*, *pin_memory=True*, *drop_last=False*, *strict_val_split=False*, **args*, ***kwargs*)

Bases: pl_bolts.datamodules.vision_datamodule.VisionDataModule

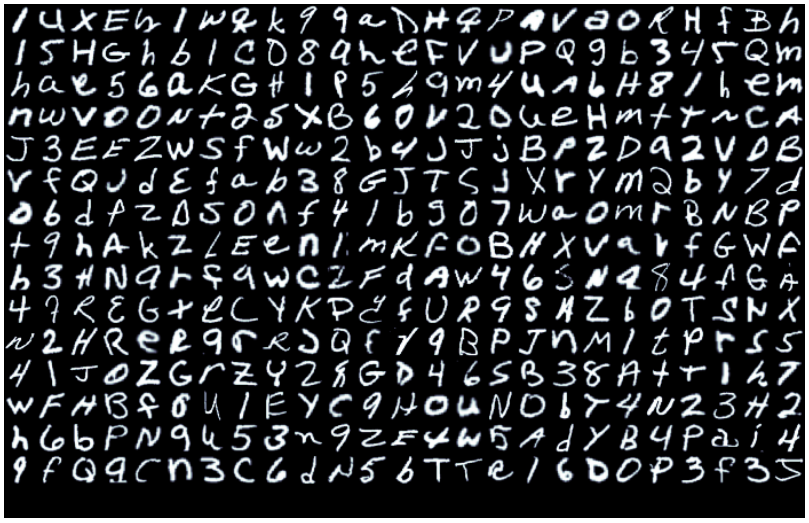


Table 1: Dataset information (source: EMNIST: an extension of MNIST to handwritten letters [Table-II])

Split Name	No. classes	Train set size	Test set size	Validation set	Total size
"byclass"	62	697,932	116,323	No	814,255
"byclass"	62	697,932	116,323	No	814,255
"bymerge"	47	697,932	116,323	No	814,255
"balanced"	47	112,800	18,800	Yes	131,600
"digits"	10	240,000	40,000	Yes	280,000
"letters"	37	88,800	14,800	Yes	103,600
"mnist"	10	60,000	10,000	Yes	70,000

Parameters

- **data_dir** (Optional[str]) – Root directory of dataset.
- **split** (str) – The dataset has 6 different splits: byclass, bymerge, balanced, letters, digits and mnist. This argument is passed to torchvision.datasets.EMNIST.
- **val_split** (Union[int, float]) – Percent (float) or number (int) of samples to use for the validation split.
- **num_workers** (int) – How many workers to use for loading data
- **normalize** (bool) – If True, applies image normalize.
- **batch_size** (int) – How many samples per batch to load.
- **seed** (int) – Random seed to be used for train/val/test splits.
- **shuffle** (bool) – If True, shuffles the train data every epoch.
- **pin_memory** (bool) – If True, the data loader will copy Tensors into CUDA pinned memory before returning them.
- **drop_last** (bool) – If True, drops the last incomplete batch.

- **strict_val_split** (bool) – If True, uses the validation split defined in the paper and ignores val_split. Note that it only works with "balanced", "digits", "letters", "mnist" splits.

Here is the default EMNIST, train, val, test-splits and transforms.

Transforms:

```
emnist_transforms = transform_lib.Compose([
    transform_lib.ToTensor(),
])
```

Example:

```
from pl_bolts.datamodules import EMNISTDataModule

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

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

Parameters

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

dataset_cls
alias of torchvision.datasets.mnist.EMNIST

default_transforms()
Default transform for the dataset.

Return type Callable

prepare_data(**args*, ***kwargs*)
Saves files to data_dir.

Return type None

setup(*stage=None*)
Creates train, val, and test dataset.

Return type None

property num_classes: int
Returns the number of classes.

See the table above.

Return type int

10.1.6 FashionMNIST

class pl_bolts.datamodules.fashion_mnist_datamodule.**FashionMNISTDataModule**(*data_dir=None*, *val_split=0.2*, *num_workers=0*, *normalize=False*, *batch_size=32*, *seed=42*, *shuffle=True*, *pin_memory=True*, *drop_last=False*, **args*, ***kwargs*)

Bases: pl_bolts.datamodules.vision_datamodule.VisionDataModule

Parameters

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

Specs:

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

Standard FashionMNIST, train, val, test splits and transforms

Transforms:

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

Example:

```
from pl_bolts.datamodules import FashionMNISTDataModule

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

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

Parameters

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

dataset_cls
alias of torchvision.datasets.mnist.FashionMNIST

default_transforms()
Default transform for the dataset.

Return type Callable

property num_classes: int
Returns the number of classes.

Return type int

10.1.7 Imagenet

class pl_bolts.datamodules.imagenet_datamodule.**ImagenetDataModule**(*data_dir*, *meta_dir=None*, *num_imgs_per_val_class=50*, *image_size=224*, *num_workers=0*, *batch_size=32*, *shuffle=True*, *pin_memory=True*, *drop_last=False*, **args*, ***kwargs*)

Bases: pytorch_lightning.core.datamodule.LightningDataModule

Warning: The feature ImagenetDataModule is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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, datamodule=dm)
```

Parameters

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

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

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

> **Warning:** Please download imagenet on your own first.

Return type None

test_dataloader()
Uses the validation split of imagenet2012 for testing.

Return type DataLoader

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

Return type DataLoader

train_transform()
The standard imagenet transforms.

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

Return type Callable

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

Parameters

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

Return type DataLoader

val_transform()
The standard imagenet transforms for validation.

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

(continues on next page)

(continued from previous page)

```
        std=[0.229, 0.224, 0.225]
    ),
])
```

Return type Callable

property num_classes: int

Return:

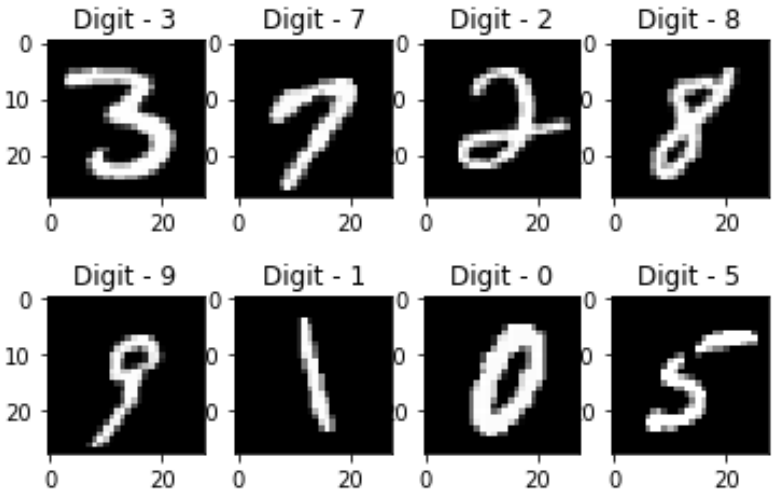
1000

Return type int

10.1.8 MNIST

class pl_bolts.datamodules.mnist_datamodule.**MNISTDataModule**(*data_dir=None*, *val_split=0.2*, *num_workers=0*, *normalize=False*, *batch_size=32*, *seed=42*, *shuffle=True*, *pin_memory=True*, *drop_last=False*, **args*, ***kwargs*)

Bases: pl_bolts.datamodules.vision_datamodule.VisionDataModule



Specs:

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

Standard MNIST, train, val, test splits and transforms

Transforms:

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

Example:

```
from pl_bolts.datamodules import MNISTDataModule

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

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

Parameters

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

dataset_cls
alias of torchvision.datasets.mnist.MNIST

default_transforms()
Default transform for the dataset.

Return type Callable

property num_classes: int
Return: 10

Return type int

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=0*, *batch_size=32*, *shuffle=True*, *pin_memory=True*, *drop_last=False*, **args*, ***kwargs*)

Bases: pytorch_lightning.core.datamodule.LightningDataModule

Warning: The feature SSLImagenetDataModule is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

prepare_data_per_node
If True, each LOCAL_RANK=0 will call prepare data. Otherwise only NODE_RANK=0, LOCAL_RANK=0 will prepare data.

allow_zero_length_dataloader_with_multiple_devices
If True, dataloader with zero length within local rank is allowed. Default value is False.

prepare_data()
Use this to download and prepare data. Downloading and saving data with multiple processes (distributed settings) will result in corrupted data. Lightning ensures this method is called only within a single process, so you can safely add your downloading logic within.

Warning: DO NOT set state to the model (use `setup` instead) since this is NOT called on every device

Example:

```
def prepare_data(self):
    # good
    download_data()
    tokenize()
    etc()

    # bad
    self.split = data_split
    self.some_state = some_other_state()
```

In a distributed environment, `prepare_data` can be called in two ways (using prepare_data_per_node)

1. Once per node. This is the default and is only called on LOCAL_RANK=0.
2. Once in total. Only called on GLOBAL_RANK=0.

Example:

```
# DEFAULT
# called once per node on LOCAL_RANK=0 of that node
```

(continues on next page)

(continued from previous page)

```
class LitDataModule(LightningDataModule):
    def __init__(self):
        super().__init__()
        self.prepare_data_per_node = True


# call on GLOBAL_RANK=0 (great for shared file systems)
class LitDataModule(LightningDataModule):
    def __init__(self):
        super().__init__()
        self.prepare_data_per_node = False
```

This is called before requesting the dataloaders:

```
model.prepare_data()
initialize_distributed()
model.setup(stage)
model.train_dataloader()
model.val_dataloader()
model.test_dataloader()
model.predict_dataloader()
```

Return type None

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

Implement one or multiple PyTorch DataLoaders for testing.

For data processing use the following pattern:

- download in *prepare_data()*
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `test()`
- *prepare_data()*
- `setup()`

Note: Lightning adds the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

Return type DataLoader

Returns A `torch.utils.data.DataLoader` or a sequence of them specifying testing samples.

Example:

```
def test_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=False, transform=transform,
                    download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=False
    )

    return loader

# can also return multiple dataloaders
def test_dataloader(self):
    return [loader_a, loader_b, ..., loader_n]
```

Note: If you don't need a test dataset and a `test_step()`, you don't need to implement this method.

Note: In the case where you return multiple test dataloaders, the `test_step()` will have an argument `dataloader_idx` which matches the order here.

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

Implement one or more PyTorch DataLoaders for training.

Return type DataLoader

Returns A collection of `torch.utils.data.DataLoader` specifying training samples. In the case of multiple dataloaders, please see this section.

The dataloader you return will not be reloaded unless you set `reload_dataloaders_every_n_epochs` to a positive integer.

For data processing use the following pattern:

- download in *prepare_data()*
- process and split in `setup()`

However, the above are only necessary for distributed processing.

Warning: do not assign state in prepare_data

- `fit()`
- *prepare_data()*
- `setup()`

Note: Lightning adds the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

Example:

```
# single dataloader
def train_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=True, transform=transform,
                    download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=True
    )
    return loader

# multiple dataloaders, return as list
def train_dataloader(self):
    mnist = MNIST(...)
    cifar = CIFAR(...)
    mnist_loader = torch.utils.data.DataLoader(
        dataset=mnist, batch_size=self.batch_size, shuffle=True
    )
    cifar_loader = torch.utils.data.DataLoader(
        dataset=cifar, batch_size=self.batch_size, shuffle=True
    )
    # each batch will be a list of tensors: [batch_mnist, batch_cifar]
    return [mnist_loader, cifar_loader]

# multiple dataloader, return as dict
def train_dataloader(self):
    mnist = MNIST(...)
    cifar = CIFAR(...)
    mnist_loader = torch.utils.data.DataLoader(
        dataset=mnist, batch_size=self.batch_size, shuffle=True
    )
    cifar_loader = torch.utils.data.DataLoader(
        dataset=cifar, batch_size=self.batch_size, shuffle=True
    )
    # each batch will be a dict of tensors: {'mnist': batch_mnist, 'cifar': batch_
↪cifar}
    return {'mnist': mnist_loader, 'cifar': cifar_loader}
```

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

Implement one or multiple PyTorch DataLoaders for validation.

The dataloader you return will not be reloaded unless you set `reload_dataloaders_every_n_epochs` to a positive integer.

It's recommended that all data downloads and preparation happen in *prepare_data()*.

- `fit()`
- `validate()`
- *prepare_data()*
- `setup()`

Note: Lightning adds the correct sampler for distributed and arbitrary hardware There is no need to set it yourself.

Return type DataLoader

Returns A torch.utils.data.DataLoader or a sequence of them specifying validation samples.

Examples:

```
def val_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=False,
                    transform=transform, download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=False
    )

    return loader

# can also return multiple dataloaders
def val_dataloader(self):
    return [loader_a, loader_b, ..., loader_n]
```

Note: If you don't need a validation dataset and a validation_step(), you don't need to implement this method.

Note: In the case where you return multiple validation dataloaders, the validation_step() will have an argument dataloader_idx which matches the order here.

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=0*, *batch_size=32*, *seed=42*, *shuffle=True*, *pin_memory=True*, *drop_last=False*, **args*, ***kwargs*)

Bases: pytorch_lightning.core.datamodule.LightningDataModule

Warning: The feature STL10DataModule is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html



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:

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

Example:

```
from pl_bolts.datamodules import STL10DataModule

dm = STL10DataModule(PATH)
model = LitModel()

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

Parameters

- **data_dir** (Optional[str]) – where to save/load the data
- **unlabeled_val_split** (int) – how many images from the unlabeled training split to use for validation
- **train_val_split** (int) – how many images from the labeled training split to use for validation
- **num_workers** (int) – how many workers to use for loading data
- **batch_size** (int) – the batch size
- **seed** (int) – random seed to be used for train/val/test splits
- **shuffle** (bool) – If true shuffles the data every epoch

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

prepare_data()
Downloads the unlabeled, train and test split.

Return type None

test_dataloader()
Loads the test split of STL10.

Parameters

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

Return type DataLoader

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

Return type DataLoader

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

Parameters

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

Return type DataLoader

val_dataloader()
Loads a portion of the 'unlabeled' training data set aside for validation.

The val dataset = (unlabeled - train_val_split)

Parameters

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

Return type DataLoader

val_dataloader_mixed()
Loads a portion of the 'unlabeled' training data set aside for validation along with the portion of the 'train' dataset to be used for validation.

unlabeled_val = (unlabeled - train_val_split)

labeled_val = (train- train_val_split)

full_val = unlabeled_val + labeled_val

Parameters

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

Return type DataLoader

CHAPTER

ELEVEN

DEBUG DATASETS

11.1 DummyDataset

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

Bases: Generic[torch.utils.data.dataset.T_co]

Generate a dummy dataset.

Example

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

Parameters

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

11.2 DummyDetectionDataset

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

Bases: Generic[torch.utils.data.dataset.T_co]

Generate a dummy dataset for object detection.

Example

```
>>> from pl_bolts.datasets import DummyDetectionDataset
>>> from torch.utils.data import DataLoader
>>> ds = DummyDetectionDataset()
>>> dl = DataLoader(ds, batch_size=7)
>>> # get first batch
>>> batch = next(iter(dl))
>>> x,y = batch
>>> x.size()
torch.Size([7, 3, 256, 256])
>>> y['boxes'].size()
torch.Size([7, 1, 4])
>>> y['labels'].size()
torch.Size([7, 1])
```

Parameters

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

11.3 RandomDataset

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

Bases: Generic[torch.utils.data.dataset.T_co]

Generate a dummy dataset.

Example

```
>>> from pl_bolts.datasets import RandomDataset
>>> from torch.utils.data import DataLoader
>>> ds = RandomDataset(10)
>>> dl = DataLoader(ds, batch_size=7)
>>> batch = next(iter(dl))
>>> len(batch),len(batch[0])
(7, 10)
```

Parameters

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

11.4 RandomDictDataset

class pl_bolts.datasets.dummy_dataset.**RandomDictDataset**(*size*, *num_samples=250*)
Bases: Generic[torch.utils.data.dataset.T_co]

Generate a dummy dataset with a dict structure.

Example

```
>>> from pl_bolts.datasets import RandomDictDataset
>>> from torch.utils.data import DataLoader
>>> ds = RandomDictDataset(10)
>>> dl = DataLoader(ds, batch_size=7)
>>> batch = next(iter(dl))
>>> len(batch['a']),len(batch['a'][0])
(7, 10)
>>> len(batch['b']),len(batch['b'][0])
(7, 10)
```

Parameters

- **size** (int) – integer representing the length of a feature_vector
- **num_samples** (int) – number of samples

11.5 RandomDictStringDataset

class pl_bolts.datasets.dummy_dataset.**RandomDictStringDataset**(*size*, *num_samples=250*)
Bases: Generic[torch.utils.data.dataset.T_co]

Generate a dummy dataset with in dict structure with strings as indexes.

Example

```
>>> from pl_bolts.datasets import RandomDictStringDataset
>>> from torch.utils.data import DataLoader
>>> ds = RandomDictStringDataset(10)
>>> dl = DataLoader(ds, batch_size=7)
>>> batch = next(iter(dl))
>>> batch['id']
['0', '1', '2', '3', '4', '5', '6']
>>> len(batch['x'])
7
```

Parameters

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

CHAPTER

TWELVE

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.

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

Example:

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

for b in dataloader:
    ...
```

class pl_bolts.datamodules.async_dataloader.**AsynchronousLoader**(*data*, *device=device(type='cuda', index=0)*, *q_size=10*, *num_batches=None*, ***kwargs*)

Bases: object

Warning: The feature AsynchronousLoader is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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

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

Parameters

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

- **`**kwargs`** – Any additional arguments to pass to the dataloader if we're constructing one here

CHAPTER

THIRTEEN

REINFORCEMENT LEARNING

These are common losses used in RL.

13.1 DQN Loss

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

> **Warning:** The feature dqn_loss is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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

13.2 Double DQN Loss

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

> **Warning:** The feature double_dqn_loss is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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

13.3 Per DQN Loss

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

> **Warning:** The feature per_dqn_loss is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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

FOURTEEN

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

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

14.1 Predicting on your data

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

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

```
from pl_bolts.models.self_supervised import SimCLR

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

# which one will be better?
```

Bolts are often trained on more than just one dataset.

```
from pl_bolts.models.self_supervised import SimCLR

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

simclr.freeze()
```

14.2 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.

14.2.1 Unfrozen Finetuning

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

```
from pl_bolts.models.self_supervised import SimCLR

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

```
classifier = LogisticRegression(...)

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

Or as a LightningModule

```
class FineTuner(pl.LightningModule):

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

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

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

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

14.2.2 Freeze then unfreeze

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

```
# freeze!
from pl_bolts.models.self_supervised import SimCLR

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

```
classifier = LogisticRegression(...)

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

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

Note: In practice, unfreezing later works MUCH better.

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

```
class UnFreezeCallback(Callback):
```

(continues on next page)

(continued from previous page)

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

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

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

```
class FineTuner(pl.LightningModule):

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

    def training_step(self, batch, batch_idx):

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

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

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

14.2.3 Hyperparameter search

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

```
from pl_bolts.models.autoencoders import VAE

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

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

14.3 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_dataloaders=train_data, val_dataloaders=val_data)
```

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

14.4 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.

14.4.1 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,
            '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
```

(continues on next page)

(continued from previous page)

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

14.4.2 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, CPC_v2

default_amdim_task = AMDIM().contrastive_task
model = CPC_v2(contrastive_task=default_amdim_task, encoder='cpc_default')
# you might need to modify the cpc encoder depending on what you use
```

14.4.3 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()
```

(continues on next page)

(continued from previous page)

```
        self.vae = VAE()
        self.amdim = AMDIM()
        self.cpc = CPC_v2

    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

FIFTEEN

AUTOENCODERS

This section houses autoencoders and variational autoencoders.

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

15.1 Basic AE

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

```
from pl_bolts.models.autoencoders import AE

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

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

```
from pl_bolts.models.autoencoders import AE

class MyAEFlavor(AE):

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

You can use the pretrained models present in bolts.

CIFAR-10 pretrained model:

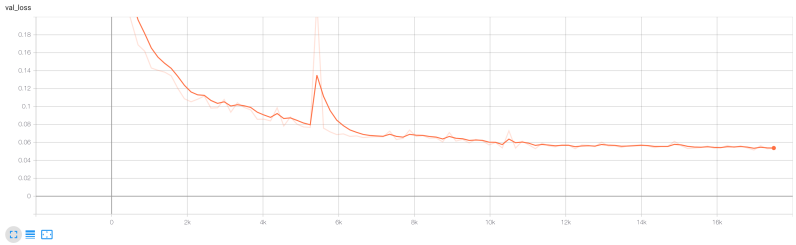
```
from pl_bolts.models.autoencoders import AE

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

ae.freeze()
```

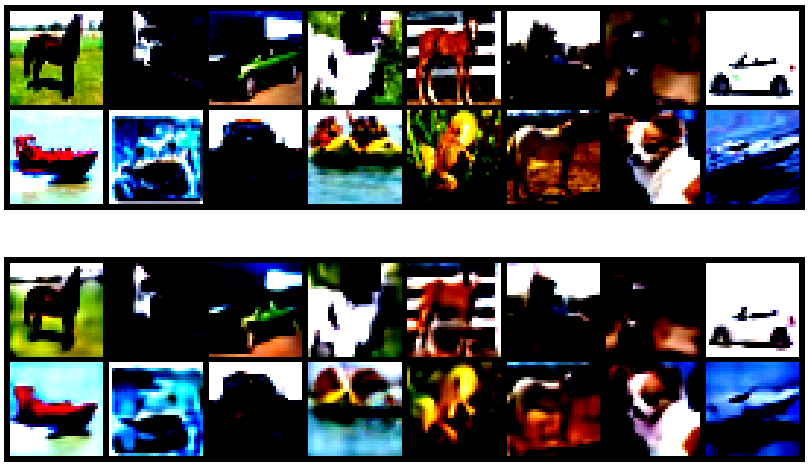
- Tensorboard for AE on CIFAR-10

Training:



Reconstructions:

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



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

Bases: pytorch_lightning.core.module.LightningModule

> **Warning:** The feature AE is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Standard AE.

Model is available pretrained on different datasets:

Example:

```
# not pretrained
ae = AE()

# pretrained on cifar10
ae = AE(input_height=32).from_pretrained('cifar10-resnet18')
```

Parameters

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

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple lr_scheduler_config).
- **Dictionary**, with an "optimizer" key, and (optionally) a "lr_scheduler" key whose value is a single LR scheduler or lr_scheduler_config.
- **Tuple of dictionaries** as described above, with an optional "frequency" key.
- **None** - Fit will run without any optimizer.

The lr_scheduler_config is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
```

(continues on next page)

(continued from previous page)

```
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency" should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track',`

metric_val) in your LightningModule.

Note: The frequency value specified in a dict along with the optimizer key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the frequency value specified in the lr_scheduler_config mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the lr_scheduler key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
```

(continues on next page)

(continued from previous page)

```
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

---

forward(*x*)

Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns Your model's output

training_step(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (`int`) – Integer displaying index of this batch
- **optimizer_idx** (`int`) – When using multiple optimizers, this argument will also be present.
- **hiddens** (`Any`) – Passed in if `truncated_bptt_steps` > 0.

Returns

Any of.

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- **`None` - Training will skip to the next batch. This is only for automatic optimization.**
  This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

---

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Returns

- Any object or value
- `None` - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)
```

(continues on next page)

(continued from previous page)

```
    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, `validation_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

15.1.1 Variational Autoencoders

15.2 Basic VAE

Use the VAE like so.

```
from pl_bolts.models.autoencoders import VAE

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

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

```
from pl_bolts.models.autoencoders import VAE

class MyVAEFlavor(VAE):

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

        return P
```

You can use the pretrained models present in bolts.

CIFAR-10 pretrained model:

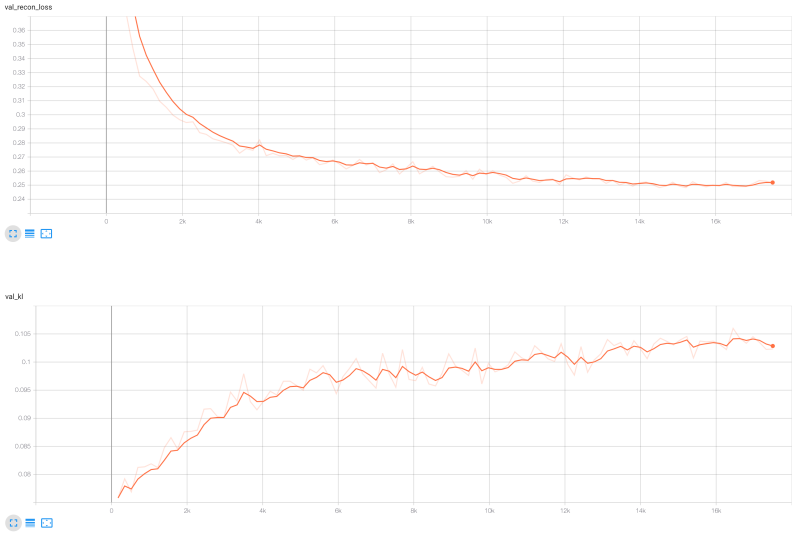
```
from pl_bolts.models.autoencoders import VAE

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

vae.freeze()
```

- Tensorboard for VAE on CIFAR-10

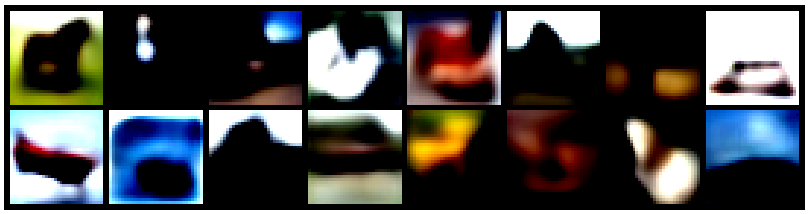
Training:



Reconstructions:

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





STL-10 pretrained model:

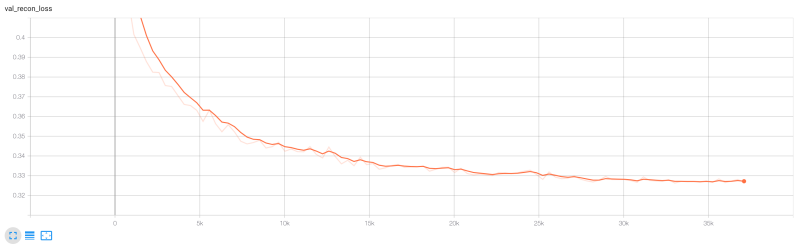
```
from pl_bolts.models.autoencoders import VAE

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

vae.freeze()
```

- Tensorboard for VAE on STL-10

Training:



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

Bases: pytorch_lightning.core.module.LightningModule

> **Warning:** The feature VAE is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Standard VAE with Gaussian Prior and approx posterior.

Model is available pretrained on different datasets:

Example:

```
# not pretrained
vae = VAE()

# pretrained on cifar10
vae = VAE(input_height=32).from_pretrained('cifar10-resnet18')

# pretrained on stl10
vae = VAE(input_height=32).from_pretrained('stl10-resnet18')
```

Parameters

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

`configure_optimizers()`

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
```

(continues on next page)

(continued from previous page)

```
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
```

(continues on next page)

(continued from previous page)

```
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

---

forward(*x*)

Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns Your model's output

training_step(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (Tensor | (Tensor, ...) | [Tensor, ...]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (int) – Integer displaying index of this batch
- **optimizer_idx** (int) – When using multiple optimizers, this argument will also be present.
- **hiddens** (Any) – Passed in if truncated_bptt_steps > 0.

Returns

Any of.

- Tensor - The loss tensor
- dict - A dictionary. Can include any keys, but must include the key 'loss'
- **None - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional optimizer_idx parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Returns

- Any object or value
- `None` - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, `validation_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

CHAPTER

SIXTEEN

CONVOLUTIONAL ARCHITECTURES

This package lists contributed convolutional architectures.

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

16.1 GPT-2

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

Bases: pytorch_lightning.core.module.LightningModule

> **Warning:** The feature GPT2 is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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.vision 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.

16.2 Image GPT

class pl_bolts.models.vision.**ImageGPT**(*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.core.module.LightningModule

> **Warning:** The feature ImageGPT is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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

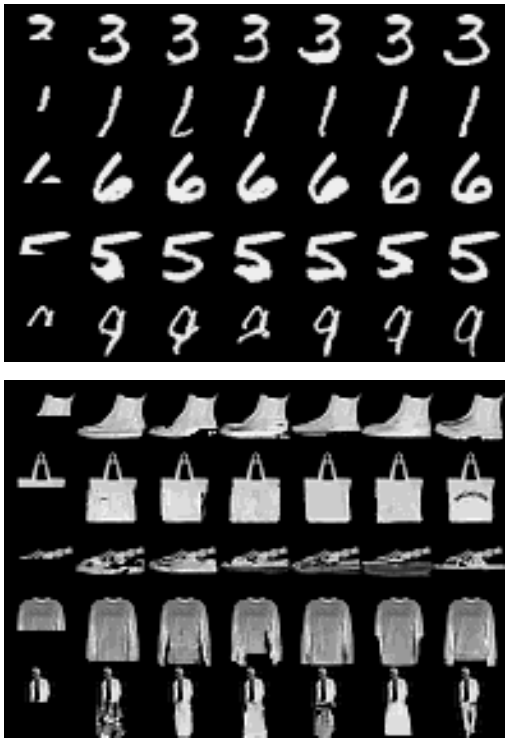
Implementation contributed by:

- 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

- **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

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
```

(continues on next page)

(continued from previous page)

```
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)
```

(continues on next page)

(continued from previous page)

```
# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule,

override the `optimizer_step()` hook.

---

forward(*x*, *classify=False*)
Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns Your model's output

test_epoch_end(*outs*)
Called at the end of a test epoch with the output of all test steps.

```
# the pseudocode for these calls
test_outs = []
for test_batch in test_data:
    out = test_step(test_batch)
    test_outs.append(out)
test_epoch_end(test_outs)
```

Parameters **outputs** – List of outputs you defined in `test_step_end()`, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader

Returns None

Note: If you didn't define a `test_step()`, this won't be called.

Examples

With a single dataloader:

```
def test_epoch_end(self, outputs):
    # do something with the outputs of all test batches
    all_test_preds = test_step_outputs.predictions

    some_result = calc_all_results(all_test_preds)
    self.log(some_result)
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each test step for that dataloader.

```
def test_epoch_end(self, outputs):
    final_value = 0
    for dataloader_outputs in outputs:
        for test_step_out in dataloader_outputs:
            # do something
            final_value += test_step_out

    self.log("final_metric", final_value)
```

test_step(*batch*, *batch_idx*)

Operates on a single batch of data from the test set. In this step you'd normally generate examples or calculate anything of interest such as accuracy.

```
# the pseudocode for these calls
test_outs = []
for test_batch in test_data:
    out = test_step(test_batch)
    test_outs.append(out)
test_epoch_end(test_outs)
```

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_id** – The index of the dataloader that produced this batch. (only if multiple test dataloaders used).

Returns

Any of.

- Any object or value
- `None` - Testing will skip to the next batch

```
# if you have one test dataloader:
def test_step(self, batch, batch_idx):
    ...


# if you have multiple test dataloaders:
def test_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single test dataset
def test_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    test_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)
```

(continues on next page)

(continued from previous page)

```
    # log the outputs!
    self.log_dict({'test_loss': loss, 'test_acc': test_acc})
```

If you pass in multiple test dataloaders, `test_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple test dataloaders
def test_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to test you don't need to implement this method.

Note: When the `test_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of the test epoch, the model goes back to training mode and gradients are enabled.

`training_step`(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (`int`) – Integer displaying index of this batch
- **optimizer_idx** (`int`) – When using multiple optimizers, this argument will also be present.
- **hiddens** (`Any`) – Passed in if `truncated_bptt_steps` > 0.

Returns

Any of.

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- **`None` - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

---

validation_epoch_end(*outs*)

Called at the end of the validation epoch with the outputs of all validation steps.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters **outputs** – List of outputs you defined in `validation_step()`, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader.

Returns None

Note: If you didn't define a `validation_step()`, this won't be called.

Examples

With a single dataloader:

```
def validation_epoch_end(self, val_step_outputs):
    for out in val_step_outputs:
        ...
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each validation step for that dataloader.

```
def validation_epoch_end(self, outputs):
    for dataloader_output_result in outputs:
        dataloader_outs = dataloader_output_result.dataloader_i_outputs

    self.log("final_metric", final_value)
```

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Returns

- Any object or value
- `None` - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...
```

(continues on next page)

(continued from previous page)

```
# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, `validation_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

16.3 Pixel CNN

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

Bases: torch.nn.modules.module.Module

> **Warning:** The feature PixelCNN is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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])
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*z*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

> **Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

16.4 UNet

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

Bases: torch.nn.modules.module.Module

Pytorch Lightning implementation of U-Net.

Paper: U-Net: Convolutional Networks for Biomedical Image Segmentation

Paper authors: Olaf Ronneberger, Philipp Fischer, Thomas Brox

Implemented by:

- Annika Brundyn
- Akshay Kulkarni

Parameters

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

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type Tensor

16.5 Semantic Segmentation

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

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

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

class pl_bolts.models.vision.**SemSegment**(*num_classes=19*, *num_layers=5*, *features_start=64*, *bilinear=False*, *ignore_index=250*, *lr=0.01*, ***kwargs*)

Bases: pytorch_lightning.core.module.LightningModule

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

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

Implemented by:

- Annika Brundyn

Example:

```
from pl_bolts.models.vision import SemSegment

model = SemSegment(num_classes=19)
dm = KittiDataModule(data_dir='/path/to/kitti/')

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

Example CLI:

```
# KITTI
python segmentation.py --data_dir /path/to/kitti/ --accelerator=gpu
```

Parameters

- **num_classes** (int) – number of output classes (default 19)
- **num_layers** (int) – number of layers in each side of U-net (default 5)
- **features_start** (int) – number of features in first layer (default 64)
- **bilinear** (bool) – whether to use bilinear interpolation (True) or transposed convolutions (default) for upsampling.
- **ignore_index** (Optional[int]) – target value to be ignored in cross_entropy (default 250)
- **lr** (float) – learning rate (default 0.01)

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple lr_scheduler_config).
- **Dictionary**, with an "optimizer" key, and (optionally) a "lr_scheduler" key whose value is a single LR scheduler or lr_scheduler_config.
- **Tuple of dictionaries** as described above, with an optional "frequency" key.
- **None** - Fit will run without any optimizer.

The lr_scheduler_config is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
```

(continues on next page)

(continued from previous page)

```
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
```

(continues on next page)

(continued from previous page)

```
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
```

(continues on next page)

(continued from previous page)

```
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

---

forward(*x*)

Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Return type `Tensor`

Returns Your model's output

training_step(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (Tensor | (Tensor, ...) | [Tensor, ...]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (int) – Integer displaying index of this batch
- **optimizer_idx** (int) – When using multiple optimizers, this argument will also be present.
- **hiddens** (Any) – Passed in if truncated_bptt_steps > 0.

Return type Dict[str, Any]

Returns

Any of.

- Tensor - The loss tensor
- dict - A dictionary. Can include any keys, but must include the key 'loss'
- **None - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional optimizer_idx parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

validation_epoch_end(*outputs*)

Called at the end of the validation epoch with the outputs of all validation steps.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters **outputs** – List of outputs you defined in `validation_step()`, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader.

Returns None

Note: If you didn't define a `validation_step()`, this won't be called.

Examples

With a single dataloader:

```
def validation_epoch_end(self, val_step_outputs):
    for out in val_step_outputs:
        ...
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each validation step for that dataloader.

```
def validation_epoch_end(self, outputs):
    for dataloader_output_result in outputs:
        dataloader_outs = dataloader_output_result.dataloader_i_outputs

    self.log("final_metric", final_value)
```

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** (`Tensor`) – The output of your `DataLoader`.

- **batch_idx** (int) – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Return type Dict[str, Any]

Returns

- Any object or value
- None - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, `validation_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

CHAPTER

SEVENTEEN

GANS

Collection of Generative Adversarial Networks

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

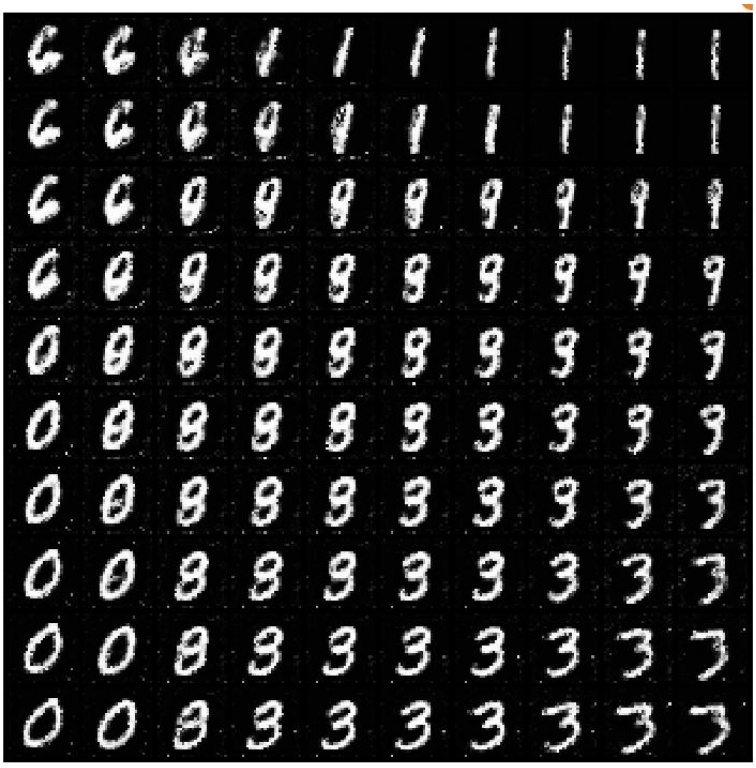
17.1 Basic GAN

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

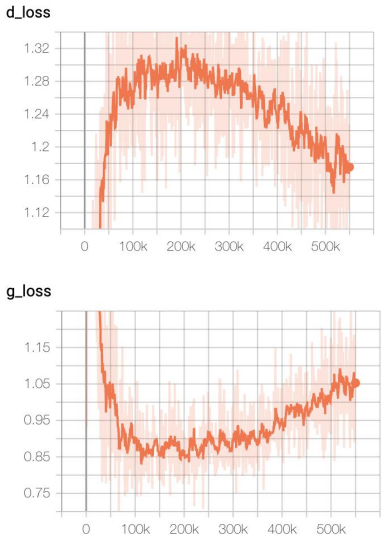
Implemented by:

- William Falcon

Example outputs:



Loss curves:




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

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

Bases: pytorch_lightning.core.module.LightningModule

Vanilla GAN implementation.

Example:

```
from pl_bolts.models.gans import GAN

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

Example CLI:

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

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

Parameters

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

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
```

(continues on next page)

(continued from previous page)

```
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
```

(continues on next page)

(continued from previous page)

```
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
```

(continues on next page)

(continued from previous page)

```
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

---

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)
```

training_step(*batch*, *batch_idx*, *optimizer_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (Tensor | (Tensor, ...) | [Tensor, ...]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (int) – Integer displaying index of this batch
- **optimizer_idx** (int) – When using multiple optimizers, this argument will also be present.
- **hiddens** (Any) – Passed in if truncated_bptt_steps > 0.

Returns

Any of.

- Tensor - The loss tensor
- dict - A dictionary. Can include any keys, but must include the key 'loss'
- **None - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional optimizer_idx parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

17.2 DCGAN

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

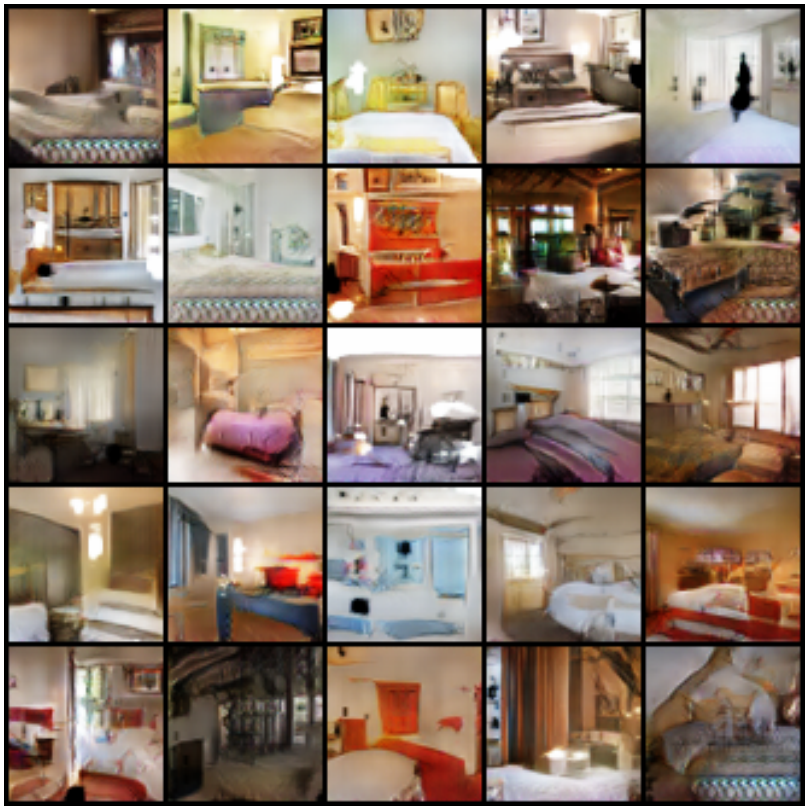
Implemented by:

- Christoph Clement

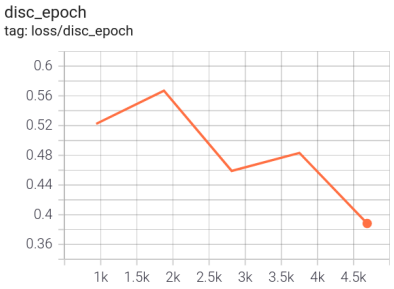
Example MNIST outputs:

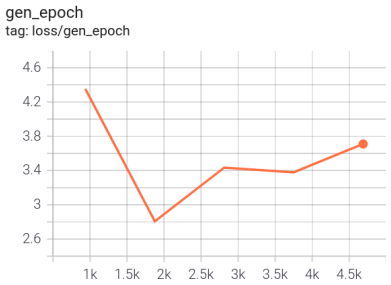


Example LSUN bedroom outputs:

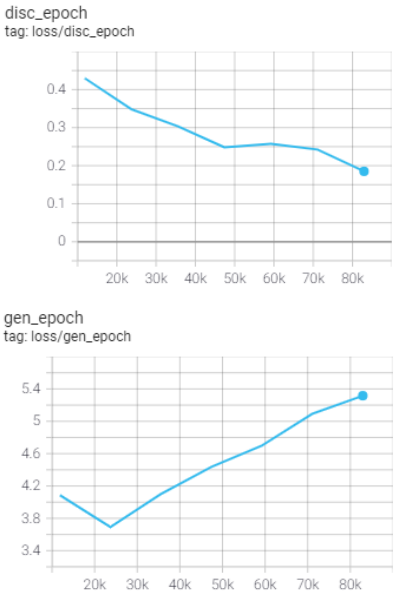


MNIST Loss curves:





LSUN Loss curves:



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

Bases: pytorch_lightning.core.module.LightningModule

DCGAN implementation.

Example:

```
from pl_bolts.models.gans import DCGAN

m = DCGAN()
Trainer(accelerator="gpu", devices=2).fit(m)
```

Example CLI:

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

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

Parameters

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

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
```

(continues on next page)

(continued from previous page)

```
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.

- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

---

forward(*noise*)

Generates an image given input noise.

Example:

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

Return type Tensor

training_step(*batch*, *batch_idx*, *optimizer_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (Tensor | (Tensor, ...) | [Tensor, ...]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (int) – Integer displaying index of this batch
- **optimizer_idx** (int) – When using multiple optimizers, this argument will also be present.
- **hiddens** (Any) – Passed in if truncated_bptt_steps > 0.

Returns

Any of.

- Tensor - The loss tensor
- dict - A dictionary. Can include any keys, but must include the key 'loss'
- **None - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional optimizer_idx parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
```

(continues on next page)

(continued from previous page)

```
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

17.3 SRGAN

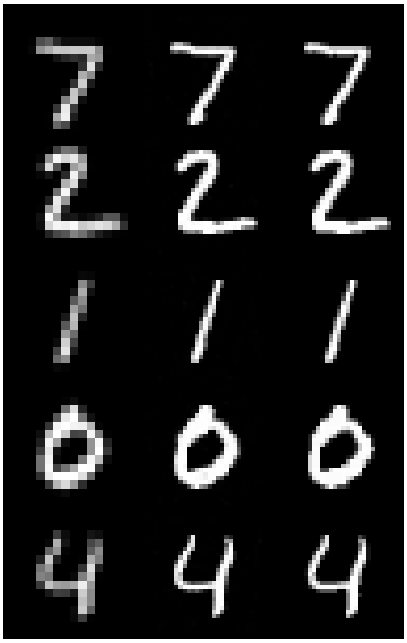
SRGAN implementation from the paper Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. The implementation is based on the version from deeplearning.ai.

Implemented by:

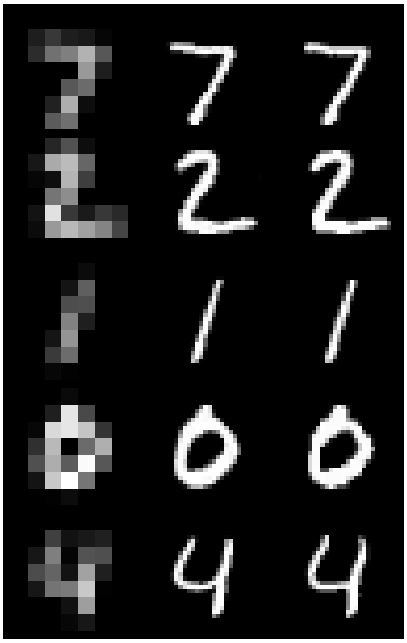
- Christoph Clement

MNIST results:

SRGAN MNIST with scale factor of 2 (left: low res, middle: generated high res, right: ground truth high res):



SRGAN MNIST with scale factor of 4:



SRResNet pretraining command used::

```
>>> python srresnet_module.py --dataset=mnist --data_dir=~/Data --scale_
↪factor=4 --save_model_checkpoint \
--batch_size=16 --num_workers=2 --gpus=4 --accelerator=ddp --precision=16 -
↪-max_steps=25000
```

SRGAN training command used::

```
>>> python srgan_module.py --dataset=mnist --data_dir=~/Data --scale_
↪factor=4 --batch_size=16 \
```

(continues on next page)

(continued from previous page)

```
--num_workers=2 --scheduler_step=29 --gpus=4 --accelerator=ddp --
↪precision=16 --max_steps=50000
```

STL10 results:

SRGAN STL10 with scale factor of 2:



SRGAN STL10 with scale factor of 4:



SRResNet pretraining command used::

```
>>>  python srresnet_module.py --dataset=stl10 --data_dir=~/Data --scale_
↪factor=4 --save_model_checkpoint \
--batch_size=16 --num_workers=2 --gpus=4 --accelerator=ddp --precision=16 -
↪-max_steps=25000
```

SRGAN training command used::

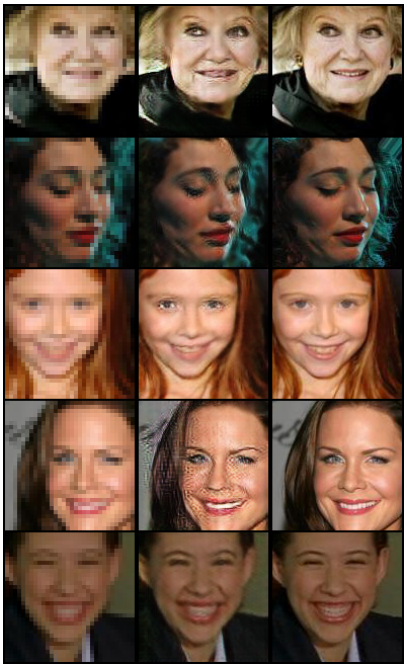
```
>>>  python srgan_module.py --dataset=stl10 --data_dir=~/Data --scale_
↪factor=4 --batch_size=16 \
--num_workers=2 --scheduler_step=29 --gpus=4 --accelerator=ddp --
↪precision=16 --max_steps=50000
```

CelebA results:

SRGAN CelebA with scale factor of 2:



SRGAN CelebA with scale factor of 4:



SRResNet pretraining command used::

```
>>> python srresnet_module.py --dataset=celeba --data_dir=~/Data --scale_
↪factor=4 --save_model_checkpoint \
--batch_size=16 --num_workers=2 --gpus=4 --accelerator=ddp --precision=16 -
↪-max_steps=25000
```

SRGAN training command used::

```
>>> python srgan_module.py --dataset=celeba --data_dir=~/Data --scale_
↪factor=4 --batch_size=16 \
--num_workers=2 --scheduler_step=29 --gpus=4 --accelerator=ddp --
↪precision=16 --max_steps=50000
```

class pl_bolts.models.gans.**SRGAN**(*image_channels=3*, *feature_maps_gen=64*, *feature_maps_disc=64*, *num_res_blocks=16*, *scale_factor=4*, *generator_checkpoint=None*, *learning_rate=0.0001*, *scheduler_step=100*, ***kwargs*)

Bases: pytorch_lightning.core.module.LightningModule

Warning: The feature SRGAN is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

SRGAN implementation from the paper Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. It uses a pretrained SRResNet model as the generator if available.

Code adapted from https-deeplearning-ai/GANs-Public to Lightning by:

- Christoph Clement

You can pretrain a SRResNet model with `srresnet_module.py`.

Example:

```
from pl_bolts.models.gan import SRGAN

m = SRGAN()
Trainer(gpus=1).fit(m)
```

Example CLI:

```
# CelebA dataset, scale_factor 4
python srgan_module.py --dataset=celeba --scale_factor=4 --gpus=1

# MNIST dataset, scale_factor 4
python srgan_module.py --dataset=mnist --scale_factor=4 --gpus=1

# STL10 dataset, scale_factor 4
python srgan_module.py --dataset=stl10 --scale_factor=4 --gpus=1
```

Parameters

- **image_channels** (int) – Number of channels of the images from the dataset
- **feature_maps_gen** (int) – Number of feature maps to use for the generator
- **feature_maps_disc** (int) – Number of feature maps to use for the discriminator
- **num_res_blocks** (int) – Number of res blocks to use in the generator
- **scale_factor** (int) – Scale factor for the images (either 2 or 4)
- **generator_checkpoint** (Optional[str]) – Generator checkpoint created with SRResNet module
- **learning_rate** (float) – Learning rate
- **scheduler_step** (int) – Number of epochs after which the learning rate gets decayed

`configure_optimizers()`

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Return type Tuple[List[Adam], List[MultiStepLR]]

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
```

(continues on next page)

(continued from previous page)

```
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]
```

(continues on next page)

(continued from previous page)

```
# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

forward(*lr_image*)

Generates a high resolution image given a low resolution image.

Example:

```
srgan = SRGAN.load_from_checkpoint(PATH)
hr_image = srgan(lr_image)
```

Return type Tensor

training_step(*batch*, *batch_idx*, *optimizer_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (Tensor | (Tensor, ...) | [Tensor, ...]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (int) – Integer displaying index of this batch
- **optimizer_idx** (int) – When using multiple optimizers, this argument will also be present.
- **hiddens** (Any) – Passed in if truncated_bptt_steps > 0.

Return type Tensor

Returns

Any of.

- Tensor - The loss tensor
- dict - A dictionary. Can include any keys, but must include the key 'loss'
- **None - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional optimizer_idx parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

class pl_bolts.models.gans.**SRResNet**(*image_channels=3*, *feature_maps=64*, *num_res_blocks=16*, *scale_factor=4*, *learning_rate=0.0001*, ***kwargs*)

Bases: `pytorch_lightning.core.module.LightningModule`

Warning: The feature SRResNet is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

SRResNet implementation from the paper Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. A pretrained SRResNet model is used as the generator for SRGAN.

Code adapted from https-deeplearning-ai/GANs-Public to Lightning by:

- Christoph Clement

Example:

```
from pl_bolts.models.gan import SRResNet

m = SRResNet()
Trainer(gpus=1).fit(m)
```

Example CLI:

```
# CelebA dataset, scale_factor 4
python srresnet_module.py --dataset=celeba --scale_factor=4 --gpus=1

# MNIST dataset, scale_factor 4
python srresnet_module.py --dataset=mnist --scale_factor=4 --gpus=1

# STL10 dataset, scale_factor 4
python srresnet_module.py --dataset=stl10 --scale_factor=4 --gpus=1
```

Parameters

- **image_channels** (`int`) – Number of channels of the images from the dataset
- **feature_maps** (`int`) – Number of feature maps to use
- **num_res_blocks** (`int`) – Number of res blocks to use in the generator
- **scale_factor** (`int`) – Scale factor for the images (either 2 or 4)
- **learning_rate** (`float`) – Learning rate

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Return type Adam

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
```

(continues on next page)

(continued from previous page)

```
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.

- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

---

forward(*lr_image*)

Creates a high resolution image given a low resolution image.

Example:

```
srresnet = SRResNet.load_from_checkpoint(PATH)
hr_image = srresnet(lr_image)
```

Return type Tensor

test_step(*batch*, *batch_idx*)

Operates on a single batch of data from the test set. In this step you'd normally generate examples or calculate anything of interest such as accuracy.

```
# the pseudocode for these calls
test_outs = []
for test_batch in test_data:
    out = test_step(test_batch)
    test_outs.append(out)
test_epoch_end(test_outs)
```

Parameters

- **batch** (Tuple[Tensor, Tensor]) – The output of your DataLoader.
- **batch_idx** (int) – The index of this batch.
- **dataloader_id** – The index of the dataloader that produced this batch. (only if multiple test dataloaders used).

Return type Tensor

Returns

Any of.

- Any object or value
- None - Testing will skip to the next batch

```
# if you have one test dataloader:
def test_step(self, batch, batch_idx):
    ...


# if you have multiple test dataloaders:
def test_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single test dataset
def test_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    test_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'test_loss': loss, 'test_acc': test_acc})
```

If you pass in multiple test dataloaders, `test_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple test dataloaders
def test_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to test you don't need to implement this method.

Note: When the `test_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of the test epoch, the model goes back to training mode and gradients are enabled.

`training_step`(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (`int`) – Integer displaying index of this batch
- **optimizer_idx** (`int`) – When using multiple optimizers, this argument will also be present.
- **hiddens** (`Any`) – Passed in if `truncated_bptt_steps` > 0.

Return type `Tensor`

Returns

Any of.

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- **`None` - Training will skip to the next batch. This is only for automatic optimization.**
  This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

---

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
```

(continues on next page)

(continued from previous page)

```
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** (Tuple[Tensor, Tensor]) – The output of your DataLoader.
- **batch_idx** (int) – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Return type Tensor

Returns

- Any object or value
- None - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)
```

(continues on next page)

(continued from previous page)

```
    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, `validation_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

CHAPTER

EIGHTEEN

OBJECT DETECTION

This package lists contributed object detection models.

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

18.1 Faster R-CNN

class pl_bolts.models.detection.faster_rcnn.faster_rcnn_module.**FasterRCNN**(*learning_rate=0.0001*, *num_classes=91*, *backbone=None*, *fpn=True*, *pretrained=False*, *pretrained_backbone=True*, *trainable_backbone_layers=3*, ***kwargs*)

Bases: pytorch_lightning.core.module.LightningModule

Warning: The feature FasterRCNN is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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_module.py --gpus 1 --pretrained True
```

Parameters

- **learning_rate** (float) – the learning rate
- **num_classes** (int) – number of detection classes (including background)
- **backbone** (Union[str, Module, None]) – Pretained backbone CNN architecture or torch.nn.Module instance.
- **fpn** (bool) – If True, creates a Feature Pyramind Network on top of Resnet based CNNs.
- **pretrained** (bool) – if true, returns a model pre-trained on COCO train2017
- **pretrained_backbone** (bool) – if true, returns a model with backbone pre-trained on Imagenet
- **trainable_backbone_layers** (int) – number of trainable resnet layers starting from final block

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple lr_scheduler_config).
- **Dictionary**, with an "optimizer" key, and (optionally) a "lr_scheduler" key whose value is a single LR scheduler or lr_scheduler_config.
- **Tuple of dictionaries** as described above, with an optional "frequency" key.
- **None** - Fit will run without any optimizer.

The lr_scheduler_config is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
```

(continues on next page)

(continued from previous page)

```
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency" should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]
```

(continues on next page)

(continued from previous page)

```
# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

---

forward(*x*)

Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns Your model's output

training_step(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (`int`) – Integer displaying index of this batch
- **optimizer_idx** (`int`) – When using multiple optimizers, this argument will also be present.
- **hiddens** (`Any`) – Passed in if `truncated_bptt_steps` > 0.

Returns

Any of.

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- **`None` - Training will skip to the next batch. This is only for automatic optimization.**
  This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

---

validation_epoch_end(*outs*)

Called at the end of the validation epoch with the outputs of all validation steps.

```
# the pseudocode for these calls
val_outs = []
```

(continues on next page)

(continued from previous page)

```
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters **outputs** – List of outputs you defined in `validation_step()`, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader.

Returns None

Note: If you didn't define a `validation_step()`, this won't be called.

Examples

With a single dataloader:

```
def validation_epoch_end(self, val_step_outputs):
    for out in val_step_outputs:
        ...
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each validation step for that dataloader.

```
def validation_epoch_end(self, outputs):
    for dataloader_output_result in outputs:
        dataloader_outs = dataloader_output_result.dataloader_i_outputs

    self.log("final_metric", final_value)
```

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Returns

- Any object or value

- None - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, `validation_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

18.2 RetinaNet

class pl_bolts.models.detection.retinanet.retinanet_module.**RetinaNet**(*learning_rate=0.0001*, *num_classes=91*, *backbone=None*, *fpn=True*, *pretrained=False*, *pretrained_backbone=True*, *trainable_backbone_layers=3*, ***kwargs*)

Bases: `pytorch_lightning.core.module.LightningModule`

Warning: The feature RetinaNet is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

PyTorch Lightning implementation of RetinaNet.

Paper: Focal Loss for Dense Object Detection.

Paper authors: Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, Piotr Dollár

Model implemented by:

- *Aditya Oke <https://github.com/oke-aditya>*

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 using LightningCLI
python retinanet_module.py --trainer.gpus 1 --model.pretrained True
```

Parameters

- **learning_rate** (`float`) – the learning rate
- **num_classes** (`int`) – number of detection classes (including background)
- **backbone** (`Optional[str]`) – Pretained backbone CNN architecture.
- **fpn** (`bool`) – If True, creates a Feature Pyramind Network on top of Resnet based CNNs.
- **pretrained** (`bool`) – if true, returns a model pre-trained on COCO train2017

- **pretrained_backbone** (bool) – if true, returns a model with backbone pre-trained on Imagenet
- **trainable_backbone_layers** (int) – number of trainable resnet layers starting from final block

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the torch.optim.lr_scheduler.ReduceLROnPlateau scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.

- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

---

forward(*x*)

Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns Your model's output

training_step(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (`int`) – Integer displaying index of this batch
- **optimizer_idx** (`int`) – When using multiple optimizers, this argument will also be present.
- **hiddens** (`Any`) – Passed in if `truncated_bptt_steps` > 0.

Returns

Any of.

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- **`None` - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

---

validation_epoch_end(*outs*)

Called at the end of the validation epoch with the outputs of all validation steps.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters **outputs** – List of outputs you defined in `validation_step()`, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader.

Returns None

Note: If you didn't define a `validation_step()`, this won't be called.

Examples

With a single dataloader:

```
def validation_epoch_end(self, val_step_outputs):
    for out in val_step_outputs:
        ...
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each validation step for that dataloader.

```
def validation_epoch_end(self, outputs):
    for dataloader_output_result in outputs:
        dataloader_outs = dataloader_output_result.dataloader_i_outputs

    self.log("final_metric", final_value)
```

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Returns

- Any object or value
- `None` - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...
```

(continues on next page)

(continued from previous page)

```
# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, `validation_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

18.3 YOLO

class pl_bolts.models.detection.yolo.yolo_module.**YOLO**(*network*, *optimizer=<class 'torch.optim.sgd.SGD'>*, *optimizer_params=None*, *lr_scheduler=<class 'pl_bolts.optimizers.lr_scheduler.LinearWarmupCosineAnnealing*, *lr_scheduler_params=None*, *confidence_threshold=0.2*, *nms_threshold=0.45*, *detections_per_image=300*)

Bases: pytorch_lightning.core.module.LightningModule

PyTorch Lightning implementation of YOLO that supports the most important features of YOLOv3, YOLOv4, YOLOv5, YOLOv7, Scaled-YOLOv4, and YOLOX.

YOLOv3 paper: Joseph Redmon and Ali Farhadi

YOLOv4 paper: Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao

YOLOv7 paper: Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao

Scaled-YOLOv4 paper: Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao

YOLOX paper: Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun

Implementation: Seppo Enarvi

The network architecture can be written in PyTorch, or read from a Darknet configuration file using the `DarknetNetwork` class. `DarknetNetwork` is also able to read weights that have been saved by Darknet. See the `CLIYOLO` command-line application for an example of how to specify a network architecture.

The input is expected to be a list of images. Each image is a tensor with shape `[channels, height, width]`. The images from a single batch will be stacked into a single tensor, so the sizes have to match. Different batches can have different image sizes, as long as the size is divisible by the ratio in which the network downsamples the input.

During training, the model expects both the image tensors and a list of targets. It's possible to train a model using one integer class label per box, but the YOLO model supports also multiple labels per box. For multi-label training, simply use a boolean matrix that indicates which classes are assigned to which boxes, in place of the class labels. *Each target is a dictionary containing the following tensors*:

- boxes (`FloatTensor[N, 4]`): the ground-truth boxes in *(x1, y1, x2, y2)* format
- labels (`Int64Tensor[N]` or `BoolTensor[N, classes]`): the class label or a boolean class mask for each ground-truth box

`forward()` method returns all predictions from all detection layers in one tensor with shape `[N, anchors, classes + 5]`, where `anchors` is the total number of anchors in all detection layers. The coordinates are scaled to the input image size. During training it also returns a dictionary containing the classification, box overlap, and confidence losses.

During inference, the model requires only the image tensor. `infer()` method filters and processes the predictions. If a prediction has a high score for more than one class, it will be duplicated. *The processed output is returned in a dictionary containing the following tensors*:

- boxes (`FloatTensor[N, 4]`): predicted bounding box *(x1, y1, x2, y2)* coordinates in image space
- scores (`FloatTensor[N]`): detection confidences
- labels (`Int64Tensor[N]`): the predicted labels for each object

Parameters

- **network** (Module) – A module that represents the network layers. This can be obtained from a Darknet configuration using `DarknetNetwork()`, or it can be defined as PyTorch code.
- **optimizer** (Type[Optimizer]) – Which optimizer class to use for training.
- **optimizer_params** (Optional[Dict[str, Any]]) – Parameters to pass to the optimizer constructor. Weight decay will be applied only to convolutional layer weights.
- **lr_scheduler** (Type[LRScheduler]) – Which learning rate scheduler class to use for training.
- **lr_scheduler_params** (Optional[Dict[str, Any]]) – Parameters to pass to the learning rate scheduler constructor.
- **confidence_threshold** (float) – Postprocessing will remove bounding boxes whose confidence score is not higher than this threshold.
- **nms_threshold** (float) – Non-maximum suppression will remove bounding boxes whose IoU with a higher confidence box is higher than this threshold, if the predicted categories are equal.
- **detections_per_image** (int) – Keep at most this number of highest-confidence detections per image.

configure_optimizers()

Constructs the optimizer and learning rate scheduler based on `self.optimizer_params` and `self.lr_scheduler_params`.

If weight decay is specified, it will be applied only to convolutional layer weights, as they contain much more parameters than the biases and batch normalization parameters. Regularizing all parameters could lead to underfitting.

Return type Tuple[List[Optimizer], List[LRScheduler]]

forward(*images*, *targets=None*)

Runs a forward pass through the network (all layers listed in `self.network`), and if training targets are provided, computes the losses from the detection layers.

Detections are concatenated from the detection layers. Each detection layer will produce a number of detections that depends on the size of the feature map and the number of anchors per feature map cell.

Parameters

- **images** (Union[Tensor, Tuple[Tensor, ...], List[Tensor]]) – A tensor of size `[batch_size, channels, height, width]` containing a batch of images or a list of image tensors.
- **targets** (Union[Tuple[Dict[str, Any], ...], List[Dict[str, Any]], None]) – If given, computes losses from detection layers against these targets. A list of target dictionaries, one for each image.

Returns Detections, and if targets were provided, a dictionary of losses. Detections are shaped `[batch_size, anchors, classes + 5]`, where `anchors` is the feature map size (width * height) times the number of anchors per cell. The predicted box coordinates are in *(x1, y1, x2, y2)* format and scaled to the input image size.

Return type detections (Tensor), losses (Tensor)

infer(*image*)

Feeds an image to the network and returns the detected bounding boxes, confidence scores, and class labels.

If a prediction has a high score for more than one class, it will be duplicated.

Parameters **image** (Tensor) – An input image, a tensor of uint8 values sized [channels, height, width].

Return type Dict[str, Any]

Returns A dictionary containing tensors "boxes", "scores", and "labels". "boxes" is a matrix of detected bounding box *(x1, y1, x2, y2)* coordinates. "scores" is a vector of confidence scores for the bounding box detections. "labels" is a vector of predicted class labels.

on_test_epoch_end()
Called in the test loop at the very end of the epoch.

Return type None

on_validation_epoch_end()
Called in the validation loop at the very end of the epoch.

Return type None

predict_step(*batch*, *batch_idx*, *dataloader_idx=0*)
Feeds a batch of images to the network and returns the detected bounding boxes, confidence scores, and class labels.

If a prediction has a high score for more than one class, it will be duplicated.

Parameters

- **batch** (Tuple[Union[Tuple[Tensor, ...], List[Tensor]], Union[Tuple[Dict[str, Any], ...], List[Dict[str, Any]]]]) – A tuple of images and targets. Images is a list of 3-dimensional tensors. Targets is a list of target dictionaries.
- **batch_idx** (int) – Index of the current batch.
- **dataloader_idx** (int) – Index of the current dataloader.

Return type List[Dict[str, Any]]

Returns A list of dictionaries containing tensors "boxes", "scores", and "labels". "boxes" is a matrix of detected bounding box *(x1, y1, x2, y2)* coordinates. "scores" is a vector of confidence scores for the bounding box detections. "labels" is a vector of predicted class labels.

process_detections(*preds*)
Splits the detection tensor returned by a forward pass into a list of prediction dictionaries, and filters them based on confidence threshold, non-maximum suppression (NMS), and maximum number of predictions.

If for any single detection there are multiple categories whose score is above the confidence threshold, the detection will be duplicated to create one detection for each category. NMS processes one category at a time, iterating over the bounding boxes in descending order of confidence score, and removes lower scoring boxes that have an IoU greater than the NMS threshold with a higher scoring box.

The returned detections are sorted by descending confidence. The items of the dictionaries are as follows: - boxes (Tensor[batch_size, N, 4]): detected bounding box *(x1, y1, x2, y2)* coordinates - scores (Tensor[batch_size, N]): detection confidences - labels (Int64Tensor[batch_size, N]): the predicted class IDs

Parameters **preds** (Tensor) – A tensor of detected bounding boxes and their attributes.

Return type List[Dict[str, Any]]

Returns Filtered detections. A list of prediction dictionaries, one for each image.

process_targets(*targets*)
Duplicates multi-label targets to create one target for each label.

Parameters **targets** (Union[Tuple[Dict[str, Any], ...], List[Dict[str, Any]]]) – List of target dictionaries. Each dictionary must contain "boxes" and "labels". "labels" is either a one-dimensional list of class IDs, or a two-dimensional boolean class map.

Return type List[Dict[str, Any]]

Returns Single-label targets. A list of target dictionaries, one for each image.

test_step(*batch*, *batch_idx*)
Evaluates a batch of data from the test set.

Parameters

- **batch** (Tuple[Union[Tuple[Tensor, ...], List[Tensor]], Union[Tuple[Dict[str, Any], ...], List[Dict[str, Any]]]]) – A tuple of images and targets. Images is a list of 3-dimensional tensors. Targets is a list of target dictionaries.
- **batch_idx** (int) – Index of the current batch.

Return type Union[Tensor, Dict[str, Any], None]

training_step(*batch*, *batch_idx*)
Computes the training loss.

Parameters

- **batch** (Tuple[Union[Tuple[Tensor, ...], List[Tensor]], Union[Tuple[Dict[str, Any], ...], List[Dict[str, Any]]]]) – A tuple of images and targets. Images is a list of 3-dimensional tensors. Targets is a list of target dictionaries.
- **batch_idx** (int) – Index of the current batch.

Return type Union[Tensor, Dict[str, Any]]

Returns A dictionary that includes the training loss in 'loss'.

validate_batch(*images*, *targets*)
Validates the format of a batch of data.

Parameters

- **images** (Union[Tensor, Tuple[Tensor, ...], List[Tensor]]) – A tensor containing a batch of images or a list of image tensors.
- **targets** (Union[Tuple[Dict[str, Any], ...], List[Dict[str, Any]], None]) – A list of target dictionaries or None. If a list is provided, there should be as many target dictionaries as there are images.

Return type None

validation_step(*batch*, *batch_idx*)
Evaluates a batch of data from the validation set.

Parameters

- **batch** (Tuple[Union[Tuple[Tensor, ...], List[Tensor]], Union[Tuple[Dict[str, Any], ...], List[Dict[str, Any]]]]) – A tuple of images and targets. Images is a list of 3-dimensional tensors. Targets is a list of target dictionaries.
- **batch_idx** (int) – Index of the current batch.

Return type Union[Tensor, Dict[str, Any], None]

CHAPTER

NINETEEN

REINFORCEMENT LEARNING

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

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

19.1 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 *accelerator=dp*. Full GPU support will be added in later updates.

19.2 DQN Models

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

19.2.1 Deep-Q-Network (DQN)

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

Original implementation by: Donal Byrne

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

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

Approximated Q Function

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

Replay Buffer

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

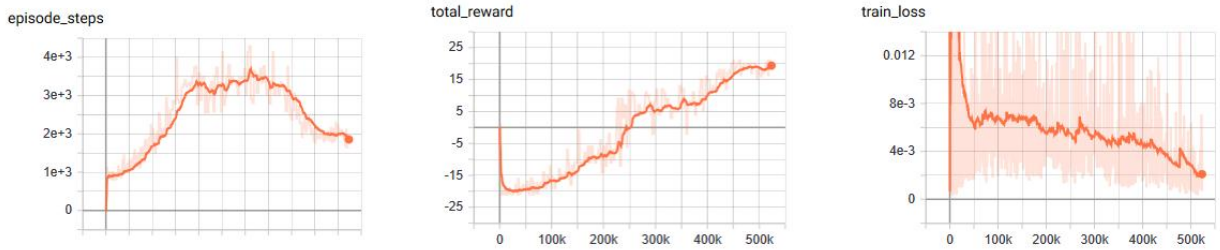
$$Q(s,a) < -(r + \gamma \max_{a' \in A} Q(s', a'))^2$$

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

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

DQN Results

DQN: Pong



Example:

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

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

Bases: `pytorch_lightning.core.module.LightningModule`

Warning: The feature DQN is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Basic DQN Model.

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

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

Example

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

Train:

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

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

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

Parameters

- **env** (`str`) – gym environment tag
- **eps_start** (`float`) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (`float`) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (`int`) – the final frame in for the decrease of epsilon. At this frame 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
- **batches_per_epoch** (int) – number of batches per epoch
- **n_steps** (int) – size of n step look ahead

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

Note: These params are fine tuned for Pong env.

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

Return type ArgumentParser

build_networks()
Initializes the DQN train and target networks.

Return type None

configure_optimizers()
Initialize Adam optimizer.

Return type List[Optimizer]

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

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

Return type Tensor

Returns q values

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

Parameters

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

Return type object

Returns gym environment

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

Return type None

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

Parameters

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

Return type List[int]

test_dataloader()
Get test loader.

Return type DataLoader

test_epoch_end(*outputs*)
Log the avg of the test results.

Return type Dict[str, Tensor]

test_step(**args*, ***kwargs*)
Evaluate the agent for 10 episodes.

Return type Dict[str, Tensor]

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

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

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

train_dataloader()
Get train loader.

Return type DataLoader

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

Parameters

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

Return type OrderedDict

Returns Training loss and log metrics

19.2.2 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

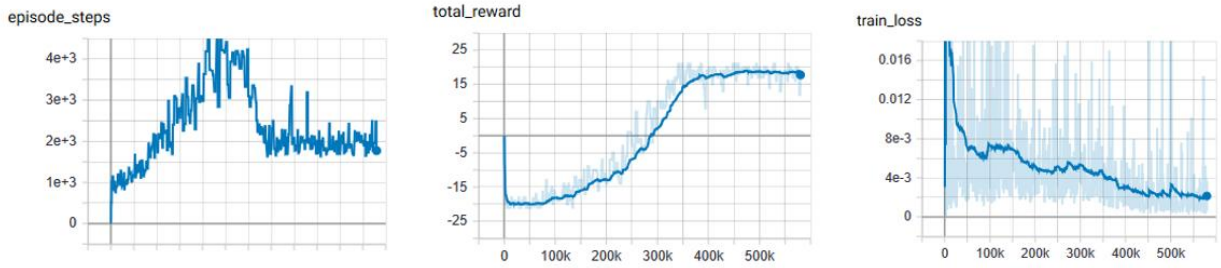
$$Q(s_t, a_t) = r_t + \gamma * \max_{Q'}(S_{t+1}, a)$$

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

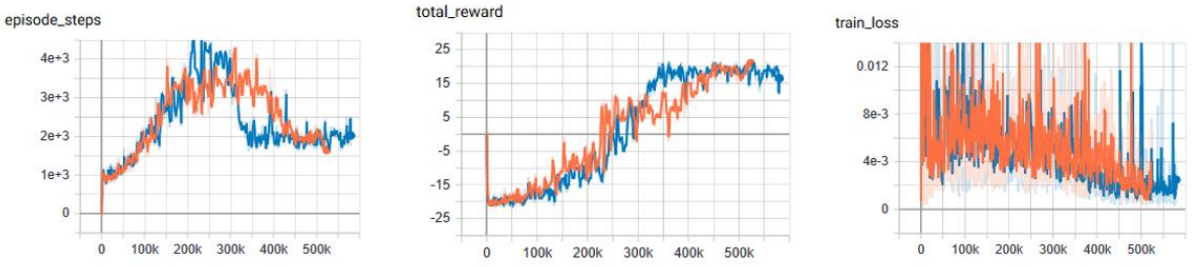
Double DQN: Pong



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

Bases: pl_bolts.models.rl.dqn_model.DQN

Warning: The feature DoubleDQN is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Double Deep Q-network (DDQN) PyTorch Lightning implementation of Double DQN.

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

Model implemented by:

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

Example

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

Train:

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

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

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

Parameters

- **env** (str) – gym environment tag
- **eps_start** (float) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (float) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (int) – the final frame in for the decrease of epsilon. At this frame 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
- **batches_per_epoch** (int) – number of batches per epoch
- **n_steps** (int) – size of n step look ahead

training_step(*batch*, _)

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

Parameters

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

Return type OrderedDict

Returns Training loss and log metrics

19.2.3 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

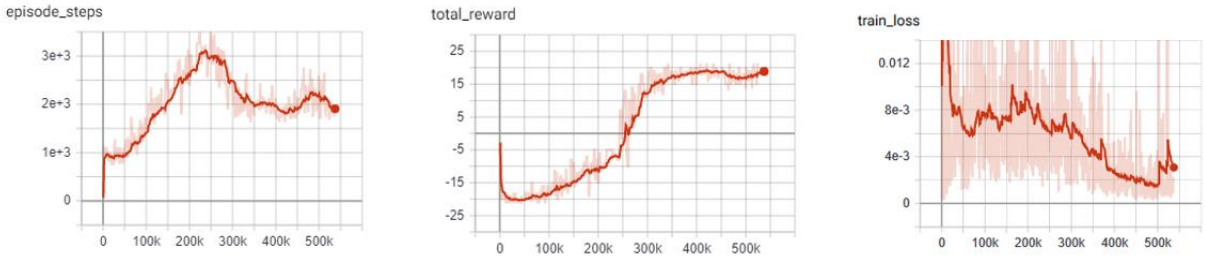
Dueling DQN Results

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.

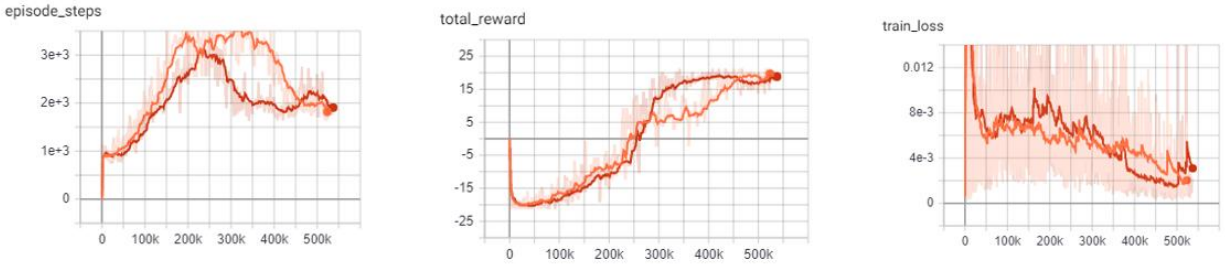


DQN vs Dueling DQN: Pong

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

Bases: pl_bolts.models.rl.dqn_model.DQN

Warning: The feature DuelingDQN is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

PyTorch Lightning implementation of Dueling DQN

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

Model implemented by:

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

Example

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

Train:

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

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

Parameters

- **env** (str) – gym environment tag

- **eps_start** (float) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (float) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (int) – the final frame in for the decrease of epsilon. At this frame 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
- **batches_per_epoch** (int) – number of batches per epoch
- **n_steps** (int) – size of n step look ahead

build_networks()
Initializes the Dueling DQN train and target networks.

Return type None

19.2.4 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 weights of the fully connect layers which get updated during backpropagation of the network. The noise parameters drive the exploration of the network instead of simply taking random actions more frequently at the start of training and less frequently towards the end. The of authors of propose two ways of doing this.

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

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

1) Independent Gaussian Noise: This injects noise per weight. For each weight a random value is taken from the distribution. Noise parameters are stored inside the layer and are updated during backpropagation. The output of the layer is calculated as normal.

2) Factorized Gaussian Noise: This injects nosier per input/ouput. In order to minimize the number of random values this method stores two random vectors, one with the size of the input and the other with the size of the output. Using these two vectors, a random matrix is generated for the layer by calculating the outer products of the vector

Noisy DQN Benefits

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

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

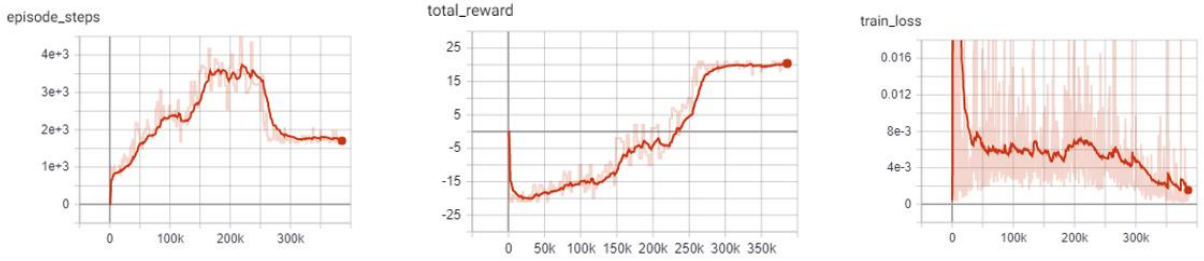
In order to update the basic DQN to a Noisy DQN we need to do the following

Noisy DQN Results

The results below improved stability and faster performance growth.

Noisy DQN baseline: Pong

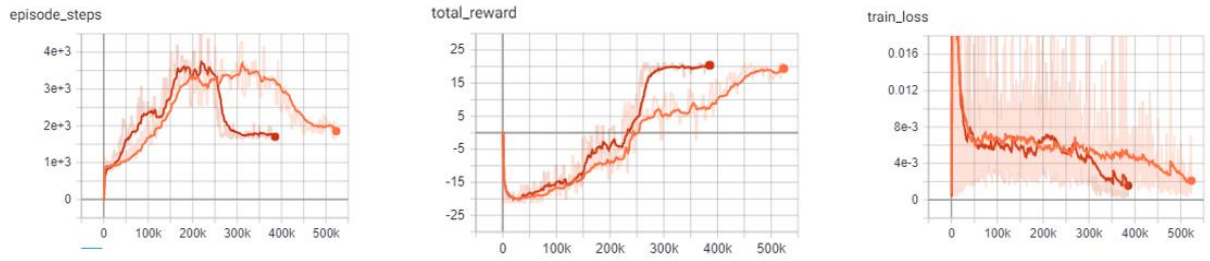
Similar to the other improvements, the average score of the agent reaches positive numbers around the 250k mark and steadily increases till convergence.



DQN vs Noisy DQN: Pong

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

- Orange: DQN
- Red: Noisy DQN



Example:

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

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

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

> **Warning:** The feature NoisyDQN is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

PyTorch Lightning implementation of Noisy DQN

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

Model implemented by:

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

Example

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

Train:

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

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

Parameters

- **env** (str) – gym environment tag
- **eps_start** (float) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (float) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (int) – the final frame in for the decrease of epsilon. At this frame 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
- **batches_per_epoch** (int) – number of batches per epoch
- **n_steps** (int) – size of n step look ahead

build_networks()
Initializes the Noisy DQN train and target networks.

Return type None

on_train_start()
Set the agents epsilon to 0 as the exploration comes from the network.

Return type None

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

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

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

19.2.5 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

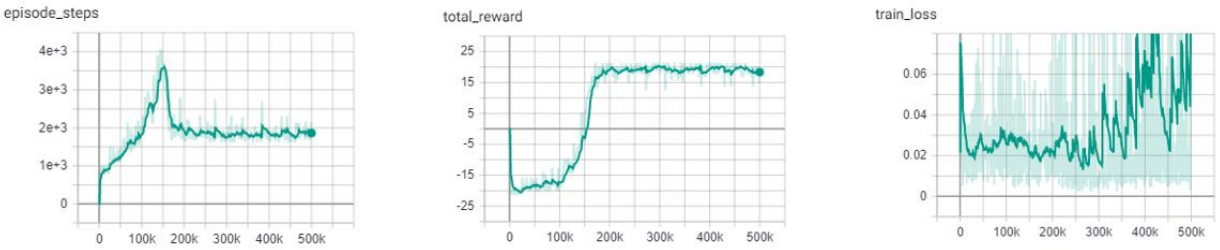
- Multi-Step learning is capable of learning faster than typical 1 step learning methods.
- Note that this method introduces a new hyperparameter n. Although n=4 is generally a good starting point and provides good results across the board.

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.

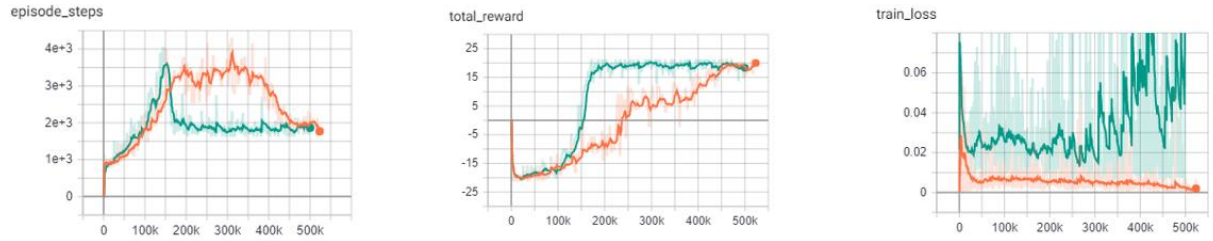
N-Step DQN: Pong

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)
```

19.2.6 Prioritized Experience Replay DQN

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

Original implementation by: Donal Byrne

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

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

The priority of each sample is defined below where

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

where 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

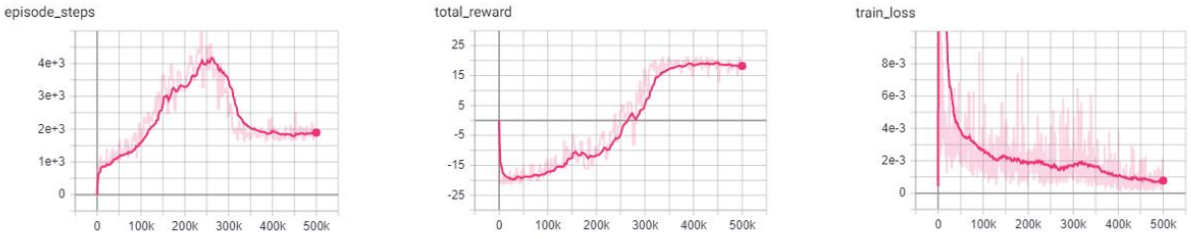
The next thing we do is to use the sample weights that we get from PER. Add the following code to the end of the loss function. This applies the weights of our sample to the batch loss. Then we return the mean loss and weighted loss for each datum, with the addition of a small epsilon value.

PER Results

The results below show improved stability and faster performance growth.

PER DQN: Pong

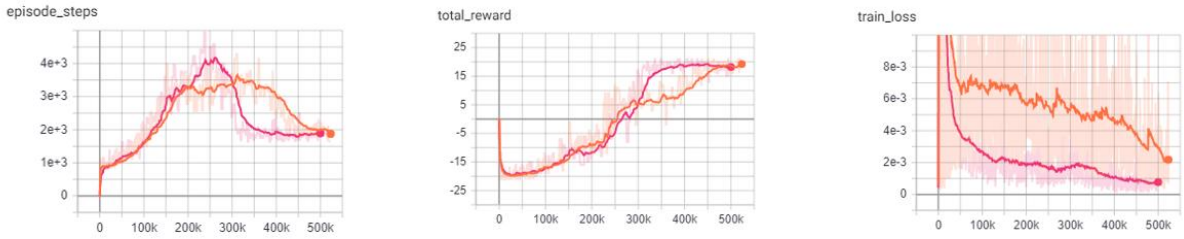
Similar to the other improvements, we see that PER improves the stability of the agents training and shows to converged on an optimal policy faster.



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

Bases: pl_bolts.models.rl.dqn_model.DQN

> **Warning:** The feature PERDQN is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

PyTorch Lightning implementation of DQN With Prioritized Experience Replay.

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

Model implemented by:

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

Example

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

Train:

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

Note:

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

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

Parameters

- **env** (str) – gym environment tag
- **eps_start** (float) – starting value of epsilon for the epsilon-greedy exploration
- **eps_end** (float) – final value of epsilon for the epsilon-greedy exploration
- **eps_last_frame** (int) – the final frame in for the decrease of epsilon. At this frame 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
- **batches_per_epoch** (int) – number of batches per epoch
- **n_steps** (int) – size of n step look ahead

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

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

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

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

Parameters

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

Return type OrderedDict

Returns Training loss and log metrics

19.3 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 continuous actions
- The policy distribution introduces stochasticity, providing natural exploration to the model
- The policy distribution provides a more stable update as a change in weights will only change the total distribution slightly, as opposed to changing weights based on the Q value of state S will change all Q values with similar states.
- Policy gradients tend to converge faste, however they are not as sample efficient and generally require more interactions with the environment.

19.3.1 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**(*env*, *gamma=0.99*, *lr=0.01*, *batch_size=8*, *n_steps=10*, *avg_reward_len=100*, *entropy_beta=0.01*, *epoch_len=1000*, *num_batch_episodes=4*, ***kwargs*)

Bases: pytorch_lightning.core.module.LightningModule

> **Warning:** The feature Reinforce is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

PyTorch Lightning implementation of REINFORCE.

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

Model implemented by:

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

Example

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

Train:

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

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

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

Parameters

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

static add_model_specific_args(*arg_parser*)

Adds arguments for DQN model.

Note: These params are fine tuned for Pong env.

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

Return type ArgumentParser

Returns arg_parser with model specific 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]

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

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

Return type float

Returns total discounted reward

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

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

Return type Tensor

Returns q values

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

Return type str

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

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

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

`train_dataloader`()
Get train loader.

Return type DataLoader

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

Parameters

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

Return type OrderedDict

Returns Training loss and log metrics

19.3.2 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.**VanillaPolicyGradient**(*env*, *gamma=0.99*, *lr=0.01*, *batch_size=8*, *n_steps=10*, *avg_reward_len=100*, *entropy_beta=0.01*, *epoch_len=1000*, ***kwargs*)

Bases: pytorch_lightning.core.module.LightningModule

> **Warning:** The feature VanillaPolicyGradient is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

PyTorch Lightning implementation of Vanilla Policy Gradient.

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

Model implemented by:

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

Example

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

Train:

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

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

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

Parameters

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

static add_model_specific_args(*arg_parser*)

Adds arguments for DQN model.

Note: These params are fine tuned for Pong env.

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

Return type ArgumentParser

Returns arg_parser with model specific cargs added

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

Parameters **rewards** – list of batched rewards

Returns list of discounted rewards

configure_optimizers()
Initialize Adam optimizer.

Return type List[Optimizer]

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

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

Return type Tensor

Returns q values

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

Return type str

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

Parameters

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

Return type Tensor

Returns loss for the current batch

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

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

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

train_dataloader()
Get train loader.

Return type DataLoader

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

Parameters

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

Return type OrderedDict

Returns Training loss and log metrics

19.4 Actor-Critic Models

The following models are based on Actor Critic. Actor Critic conbines the approaches of value-based learning (the DQN family) and the policy-based learning (the PG family) by learning the value function as well as the policy distribution. This approach updates the policy network according to the policy gradient, and updates the value network to fit the discounted rewards.

Actor Critic Key Points:

- Actor outputs a distribution of actions for controlling the agent
- Critic outputs a value of current state for policy update suggestion
- The addition of critic allows the model to do n-step training instead of generating an entire trajectory

19.4.1 Advantage Actor Critic (A2C)

(Asynchronous) Advantage Actor Critic model introduced in Asynchronous Methods for Deep Reinforcement Learning
Paper authors: Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu

Original implementation by: Jason Wang

Advantage Actor Critic (A2C) is the classical actor critic approach in reinforcement learning. The underlying neural network has an actor head and a critic head to output action distribution as well as value of current state. Usually the first few layers are shared by the two heads to prevent learning similar stuff twice. It builds upon the idea of using a baseline of average reward to reduce variance (in VPG) by using the critic as a baseline which could theoretically have better performance.

The algorithm can use an n-step training approach instead of generating an entire trajectory. The algorithm is as follows:

1. Initialize our network.
2. Rollout n steps and save the transitions (states, actions, rewards, values, dones).
3. Calculate the n-step (discounted) return by bootstrapping the last value.

$$G_{n+1} = V_{n+1}, G_t = r_t + \gamma G_{t+1} \; \forall t \in [0, n]$$

4. Calculate actor loss using values as baseline.

$$L_{actor} = -\frac{1}{n} \sum_t (G_t - V_t) \log \pi(a_t|s_t)$$

5. Calculate critic loss using returns as target.

$$L_{critic} = \frac{1}{n} \sum_t (V_t - G_t)^2$$

6. Calculate entropy bonus to encourage exploration.

$$H_\pi = -\frac{1}{n} \sum_t \pi(a_t|s_t) \log \pi(a_t|s_t)$$

7. Calculate total loss as a weighted sum of the three components above.

$$L = L_{actor} + \beta_{critic} L_{critic} - \beta_{entropy} H_\pi$$

8. Perform gradient descent to update our network.

Note: The current implementation only support discrete action space, and has only been tested on the CartPole environment.

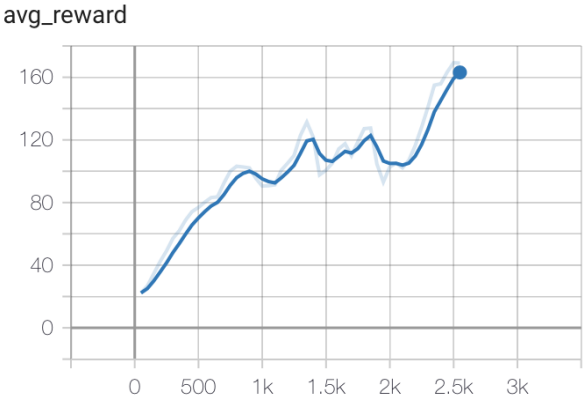
A2C Benefits

- Combines the benefit from value-based learning and policy-based learning
- Further reduces variance using the critic as a value estimator

A2C Results

Hyperparameters:

- Batch Size: 32
- Learning Rate: 0.001
- Entropy Beta: 0.01
- Critic Beta: 0.5
- Gamma: 0.99



Example:

```
from pl_bolts.models.rl import AdvantageActorCritic
a2c = AdvantageActorCritic("CartPole-v0")
trainer = Trainer()
trainer.fit(a2c)
```

class pl_bolts.models.rl.**AdvantageActorCritic**(*env*, *gamma=0.99*, *lr=0.001*, *batch_size=32*, *avg_reward_len=100*, *entropy_beta=0.01*, *critic_beta=0.5*, *epoch_len=1000*, ***kwargs*)

Bases: pytorch_lightning.core.module.LightningModule

> **Warning:** The feature AdvantageActorCritic is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

PyTorch Lightning implementation of Advantage Actor Critic.

Paper Authors: Volodymyr Mnih, Adrià Puigdomènech Badia, et al.

Model implemented by:

- Jason Wang

Example

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

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
- **avg_reward_len** (int) – how many episodes to take into account when calculating the avg reward
- **entropy_beta** (float) – dictates the level of entropy per batch
- **critic_beta** (float) – dictates the level of critic loss per batch
- **epoch_len** (int) – how many batches before pseudo epoch

static add_model_specific_args(*arg_parser*)

Adds arguments for A2C model.

Parameters **arg_parser** (ArgumentParser) – the current argument parser to add to

Return type ArgumentParser

Returns arg_parser with model specific cargs added

compute_returns(*rewards*, *dones*, *last_value*)

Calculate the discounted rewards of the batched rewards.

Parameters

- **rewards** (List[float]) – list of rewards
- **dones** (List[bool]) – list of done masks
- **last_value** (Tensor) – the predicted value for the last state (for bootstrap)

Return type Tensor

Returns tensor 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 log prob of each action and the value for the state as an output.

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

Return type Tuple[Tensor, Tensor]

Returns action log probabilities, values

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

Return type str

`loss`(*states*, *actions*, *returns*)
Calculates the loss for A2C which is a weighted sum of actor loss (MSE), critic loss (PG), and entropy (for exploration)

Parameters

- **states** (Tensor) – tensor of shape (batch_size, state dimension)
- **actions** (Tensor) – tensor of shape (batch_size,)
- **returns** (Tensor) – tensor of shape (batch_size,)

Return type Tensor

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

Return type Iterator[Tuple[ndarray, int, Tensor]]

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

Note: This is what's taken by the dataloader: states: a list of numpy array actions: a list of list of int returns: a torch tensor

---

`train_dataloader`()
Get train loader.

Return type DataLoader

`training_step`(*batch*, *batch_idx*)
Perform one actor-critic update using a batch of data.

Parameters **batch** (Tuple[Tensor, Tensor]) – a batch of (states, actions, returns)

Return type OrderedDict

19.4.2 Soft Actor Critic (SAC)

Soft Actor Critic model introduced in Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor Paper authors: Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, Sergey Levine

Original implementation by: Jason Wang

Soft Actor Critic (SAC) is a powerful actor critic algorithm in reinforcement learning. Unlike A2C, SAC's policy outputs a special continuous distribution for actions, and its critic estimates the Q value instead of the state value, which means it now takes in not only states but also actions. The new actor allows SAC to support continuous action tasks such as controlling robots, and the new critic allows SAC to support off-policy learning which is more sample efficient.

The actor has a new objective to maximize entropy to encourage exploration while maximizing the expected rewards. The critic uses two separate Q functions to "mitigate positive bias" during training by picking the minimum of the two as the predicted Q value.

Since SAC is off-policy, its algorithm's training step is quite similar to DQN:

1. Initialize one policy network, two Q networks, and two corresponding target Q networks.
2. Run 1 step using action sampled from policy and store the transition into the replay buffer.

$$a \sim tanh(N(\mu_\pi(s), \sigma_\pi(s)))$$

3. Sample transitions (states, actions, rewards, dones, next states) from the replay buffer.

$$s, a, r, d, s' \sim B$$

4. Compute actor loss and update policy network.

$$J_\pi = \frac{1}{n}\sum_i (\log \pi(\pi(a|s_i)|s_i) - Q_{min}(s_i, \pi(a|s_i)))$$

5. Compute Q target

$$target_i = r_i + (1 - d_i)\gamma(\min_i Q_{target,i}(s'_i, \pi(a', s'_i)) - log\pi(\pi(a|s'_i)|s'_i))$$

5. Compute critic loss and update Q network..

$$J_{Q_i} = \frac{1}{n}\sum_i (Q_i(s_i, a_i) - target_i)^2$$

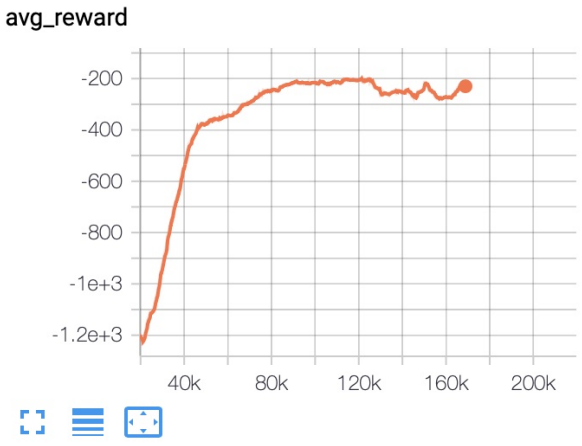
4. Soft update the target Q network using a weighted sum of itself and the Q network.

$$Q_{target,i} := \tau Q_{target,i} + (1 - \tau)Q_i$$

SAC Benefits

- More sample efficient due to off-policy training
- Supports continuous action space

SAC Results



Example:: from pl_bolts.models.rl import SAC sac = SAC("Pendulum-v0") trainer = Trainer() trainer.fit(sac)

class pl_bolts.models.rl.**SAC**(*env*, *eps_start=1.0*, *eps_end=0.02*, *eps_last_frame=150000*, *sync_rate=1*, *gamma=0.99*, *policy_learning_rate=0.0003*, *q_learning_rate=0.0003*, *target_alpha=0.005*, *batch_size=128*, *replay_size=1000000*, *warm_start_size=10000*, *avg_reward_len=100*, *min_episode_reward=- 21*, *seed=123*, *batches_per_epoch=10000*, *n_steps=1*, ***kwargs*)

Bases: pytorch_lightning.core.module.LightningModule

Warning: The feature SAC is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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

Note: These params are fine tuned for Pong env.

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

Return type ArgumentParser

build_networks()
Initializes the SAC policy and q networks (with targets)

Return type None

configure_optimizers()
Initialize Adam optimizer.

Return type Tuple[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

loss(*batch*)
Calculates the loss for SAC which contains a total of 3 losses.

Parameters **batch** (Tuple[Tensor, Tensor, Tensor, Tensor, Tensor]) – a batch of states, actions, rewards, dones, and next states

Return type Tuple[Tensor, Tensor, Tensor]

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

Return type None

run_n_episodes(*env*, *n_epsiodes=1*)
Carries out N episodes of the environment with the current agent without exploration.

Parameters

- **env** – environment to use, either train environment or test environment
- **n_epsiodes** (int) – number of episodes to run

Return type List[int]

soft_update_target(*q_net*, *target_net*)
Update the weights in target network using a weighted sum.

w_target := (1-a) * w_target + a * w_q

Parameters

- **q_net** – the critic (q) network
- **target_net** – the target (q) network

test_dataloader()
Get test loader.

Return type DataLoader

test_epoch_end(*outputs*)
Log the avg of the test results.

Return type Dict[str, Tensor]

test_step(**args*, ***kwargs*)
Evaluate the agent for 10 episodes.

Return type Dict[str, Tensor]

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

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

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

train_dataloader()
Get train loader.

Return type DataLoader

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

Parameters

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

CHAPTER

TWENTY

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

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

20.1 Use cases

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

20.1.1 Extracting image features

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

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

```
from pl_bolts.models.self_supervised import SimCLR

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

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

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

Example:

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

20.1.2 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.transforms.self_supervised.simclr_transforms 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),
)
```

20.1.3 Research

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

```
from pl_bolts.models.self_supervised import CPC_v2
from pl_bolts.losses.self_supervised_learning import FeatureMapContrastiveTask

amdim_task = FeatureMapContrastiveTask(comparisons='01, 11, 02', bidirectional=True)
model = CPC_v2(contrastive_task=amdim_task)
```

20.2 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.

20.2.1 AMDIM

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

Bases: pytorch_lightning.core.module.LightningModule

Warning: The feature AMDIM is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the torch.optim.lr_scheduler.ReduceLROnPlateau scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.

- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

---

forward(*img_1*, *img_2*)

Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns Your model's output

train_dataloader()

Implement one or more PyTorch DataLoaders for training.

Returns A collection of `torch.utils.data.DataLoader` specifying training samples. In the case of multiple dataloaders, please see this section.

The dataloader you return will not be reloaded unless you set `reload_dataloaders_every_n_epochs` to a positive integer.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in `setup()`

However, the above are only necessary for distributed processing.

> **Warning:** do not assign state in prepare_data

- `fit()`
- `prepare_data()`
- `setup()`

Note: Lightning adds the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

Example:

```
# single dataloader
def train_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=True, transform=transform,
```

(continues on next page)

(continued from previous page)

```
                    download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=True
    )
    return loader

# multiple dataloaders, return as list
def train_dataloader(self):
    mnist = MNIST(...)
    cifar = CIFAR(...)
    mnist_loader = torch.utils.data.DataLoader(
        dataset=mnist, batch_size=self.batch_size, shuffle=True
    )
    cifar_loader = torch.utils.data.DataLoader(
        dataset=cifar, batch_size=self.batch_size, shuffle=True
    )
    # each batch will be a list of tensors: [batch_mnist, batch_cifar]
    return [mnist_loader, cifar_loader]

# multiple dataloader, return as dict
def train_dataloader(self):
    mnist = MNIST(...)
    cifar = CIFAR(...)
    mnist_loader = torch.utils.data.DataLoader(
        dataset=mnist, batch_size=self.batch_size, shuffle=True
    )
    cifar_loader = torch.utils.data.DataLoader(
        dataset=cifar, batch_size=self.batch_size, shuffle=True
    )
    # each batch will be a dict of tensors: {'mnist': batch_mnist, 'cifar': batch_
↪cifar}
    return {'mnist': mnist_loader, 'cifar': cifar_loader}
```

training_step(*batch*, *batch_nb*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (Tensor | (Tensor, ...) | [Tensor, ...]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (int) – Integer displaying index of this batch
- **optimizer_idx** (int) – When using multiple optimizers, this argument will also be present.
- **hiddens** (Any) – Passed in if truncated_bptt_steps > 0.

Returns

Any of.

- Tensor - The loss tensor

- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- **`None` - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

`training_step_end`(*outputs*)

Use this when training with dp because `training_step()` will operate on only part of the batch. However, this is still optional and only needed for things like softmax or NCE loss.

Note: If you later switch to ddp or some other mode, this will still be called so that you don't have to change your code

---

```
# pseudocode
sub_batches = split_batches_for_dp(batch)
step_output = [training_step(sub_batch) for sub_batch in sub_batches]
training_step_end(step_output)
```

Parameters **step_output** – What you return in *training_step* for each batch part.

Returns Anything

When using the DP strategy, only a portion of the batch is inside the training_step:

```
def training_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self(x)

    # softmax uses only a portion of the batch in the denominator
    loss = self.softmax(out)
    loss = nce_loss(loss)
    return loss
```

If you wish to do something with all the parts of the batch, then use this method to do it:

```
def training_step(self, batch, batch_idx):
    # batch is 1/num_gpus big
    x, y = batch

    out = self.encoder(x)
    return {"pred": out}


def training_step_end(self, training_step_outputs):
    gpu_0_pred = training_step_outputs[0]["pred"]
    gpu_1_pred = training_step_outputs[1]["pred"]
    gpu_n_pred = training_step_outputs[n]["pred"]

    # this softmax now uses the full batch
    loss = nce_loss([gpu_0_pred, gpu_1_pred, gpu_n_pred])
    return loss
```

See also:

See the Multi GPU Training guide for more details.

val_dataloader()

Implement one or multiple PyTorch DataLoaders for validation.

The dataloader you return will not be reloaded unless you set `reload_dataloaders_every_n_epochs` to a positive integer.

It's recommended that all data downloads and preparation happen in `prepare_data()`.

- `fit()`
- `validate()`

- prepare_data()
- setup()

Note: Lightning adds the correct sampler for distributed and arbitrary hardware There is no need to set it yourself.

Returns A torch.utils.data.DataLoader or a sequence of them specifying validation samples.

Examples:

```
def val_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=False,
                    transform=transform, download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=False
    )

    return loader

# can also return multiple dataloaders
def val_dataloader(self):
    return [loader_a, loader_b, ..., loader_n]
```

Note: If you don't need a validation dataset and a validation_step(), you don't need to implement this method.

Note: In the case where you return multiple validation dataloaders, the validation_step() will have an argument dataloader_idx which matches the order here.

---

validation_epoch_end(*outputs*)

Called at the end of the validation epoch with the outputs of all validation steps.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters **outputs** – List of outputs you defined in validation_step(), or if there are multiple dataloaders, a list containing a list of outputs for each dataloader.

Returns None

Note: If you didn't define a `validation_step()`, this won't be called.

Examples

With a single dataloader:

```
def validation_epoch_end(self, val_step_outputs):
    for out in val_step_outputs:
        ...
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each validation step for that dataloader.

```
def validation_epoch_end(self, outputs):
    for dataloader_output_result in outputs:
        dataloader_outs = dataloader_output_result.dataloader_i_outputs

    self.log("final_metric", final_value)
```

validation_step(*batch*, *batch_nb*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Returns

- Any object or value
- `None` - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, `validation_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

20.2.2 BYOL

class pl_bolts.models.self_supervised.**BYOL**(*learning_rate=0.2*, *weight_decay=1.5e-06*, *warmup_epochs=10*, *max_epochs=1000*, *base_encoder='resnet50'*, *encoder_out_dim=2048*, *projector_hidden_dim=4096*, *projector_out_dim=256*, *initial_tau=0.996*, ***kwargs*)

Bases: pytorch_lightning.core.module.LightningModule

PyTorch Lightning implementation of Bootstrap Your Own Latent (BYOL)_

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

Parameters

- **learning_rate** (*float, optional*) – optimizer learning rate. Defaults to 0.2.
- **weight_decay** (*float, optional*) – optimizer weight decay. Defaults to 1.5e-6.
- **warmup_epochs** (*int, optional*) – number of epochs for scheduler warmup. Defaults to 10.
- **max_epochs** (*int, optional*) – maximum number of epochs for scheduler. Defaults to 1000.
- **base_encoder** (*Union[str, torch.nn.Module], optional*) – base encoder architecture. Defaults to "resnet50".
- **encoder_out_dim** (*int, optional*) – base encoder output dimension. Defaults to 2048.
- **projector_hidden_dim** (*int, optional*) – projector MLP hidden dimension. Defaults to 4096.
- **projector_out_dim** (*int, optional*) – projector MLP output dimension. Defaults to 256.
- **initial_tau** (*float, optional*) – initial value of target decay rate used. Defaults to 0.996.

Model implemented by:

- Annika Brundyn

Example:

```
model = BYOL(num_classes=10)

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

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

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
```

calculate_loss(*v_online*, *v_target*)

Calculates similarity loss between the online network prediction of target network projection.

Parameters

- **v_online** (*Tensor*) – Online network view
- **v_target** (*Tensor*) – Target network view

Return type Tensor

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
```

(continues on next page)

(continued from previous page)

```
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to

the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
```

(continues on next page)

(continued from previous page)

```
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

forward(*x*)

Returns the encoded representation of a view.

Parameters **x** (*Tensor*) – sample to be encoded

Return type Tensor

on_train_batch_end(*outputs*, *batch*, *batch_idx*)

Add callback to perform exponential moving average weight update on target network.

Return type None

training_step(*batch*, *batch_idx*)

Complete training loop.

Return type Tensor

validation_step(*batch*, *batch_idx*)

Complete validation loop.

Return type Tensor

20.2.3 CPC (V2)

PyTorch Lightning implementation of Data-Efficient Image Recognition with Contrastive Predictive Coding

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

Model implemented by:

- William Falcon
- Tullie Murrell

To Train:

```
import pytorch_lightning as pl
from pl_bolts.models.self_supervised import CPC_v2
from pl_bolts.datamodules import CIFAR10DataModule
from pl_bolts.transforms.self_supervised.cpc_transforms import (
    CPCTrainTransformsCIFAR10,
    CPCEvalTransformsCIFAR10
)

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

# model
model = CPC_v2()

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

To finetune:

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

CIFAR-10 and STL-10 baselines

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

Table 1: CPCv2 implementation results

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

CIFAR-10 pretrained model:

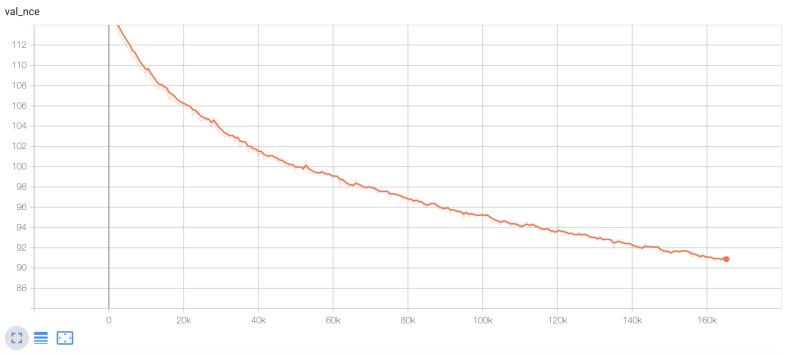
```
from pl_bolts.models.self_supervised import CPC_v2

weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/cpc/cpc-cifar10-v4-
↪exp3/epoch%3D474.ckpt'
cpc_v2 = CPC_v2.load_from_checkpoint(weight_path, strict=False)

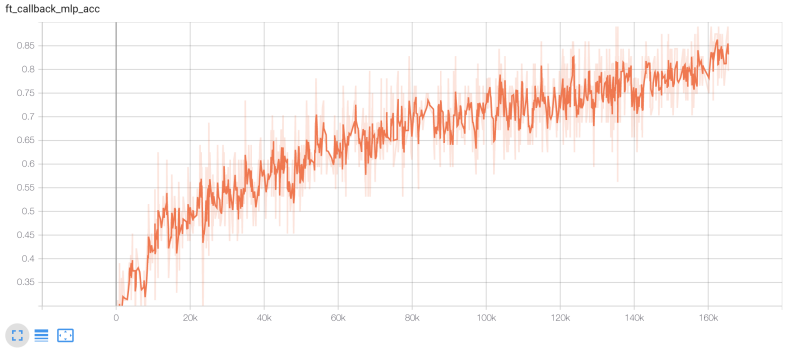
cpc_v2.freeze()
```

- Tensorboard for CIFAR10

Pre-training:



Fine-tuning:



STL-10 pretrained model:

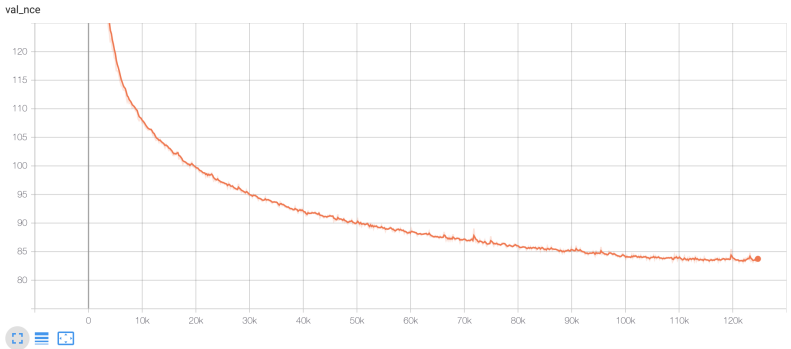
```
from pl_bolts.models.self_supervised import CPC_v2

weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/cpc/cpc-stl10-v0-exp3/
↪epoch%3D624.ckpt'
cpc_v2 = CPC_v2.load_from_checkpoint(weight_path, strict=False)

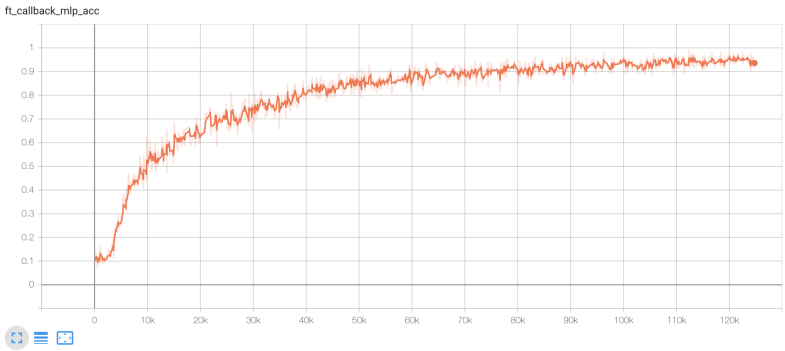
cpc_v2.freeze()
```

- Tensorboard for STL10

Pre-training:



Fine-tuning:



20.2.4 CPC (v2) API

class pl_bolts.models.self_supervised.**CPC_v2**(*encoder_name='cpc_encoder'*, *patch_size=8*, *patch_overlap=4*, *online_ft=True*, *task='cpc'*, *num_workers=4*, *num_classes=10*, *learning_rate=0.0001*, *pretrained=None*, ***kwargs*)

Bases: pytorch_lightning.core.module.LightningModule

> **Warning:** The feature CPC_v2 is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Parameters

- **encoder_name** (str) – 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 each patch should have
- **online_ft** (bool) – If True, enables a 1024-unit MLP to fine-tune online
- **task** (str) – Which self-supervised task to use ('cpc', 'amdim', etc…)
- **num_workers** (int) – number of dataloader workers
- **num_classes** (int) – number of classes
- **learning_rate** (float) – learning rate
- **pretrained** (Optional[str]) – If true, will use the weights pretrained (using CPC) on Imagenet

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.

- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
```

(continues on next page)

(continued from previous page)

```
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
```

(continues on next page)

(continued from previous page)

```
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

forward(*img*)

Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns Your model's output

training_step(*batch*, *batch_nb*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (`int`) – Integer displaying index of this batch
- **optimizer_idx** (`int`) – When using multiple optimizers, this argument will also be present.
- **hiddens** (`Any`) – Passed in if `truncated_bptt_steps` > 0.

Returns

Any of.

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- **`None` - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

---

validation_step(*batch*, *batch_nb*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Returns

- Any object or value
- `None` - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, `validation_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

20.2.5 Moco (v2) API

class pl_bolts.models.self_supervised.**Moco_v2**(*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*, *data_dir='./'*, *batch_size=256*, *use_mlp=False*, *num_workers=8*, **args*, ***kwargs*)

Bases: pytorch_lightning.core.module.LightningModule

Warning: The feature Moco_v2 is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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 Moco_v2
model = Moco_v2()
trainer = Trainer()
trainer.fit(model)
```

CLI command:

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

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

Parameters

- **base_encoder** (Union[str, Module]) – torchvision model name or torch.nn.Module
- **emb_dim** (int) – feature dimension (default: 128)
- **num_negatives** (int) – queue size; number of negative keys (default: 65536)
- **encoder_momentum** (float) – moco momentum of updating key encoder (default: 0.999)
- **softmax_temperature** (float) – softmax temperature (default: 0.07)
- **learning_rate** (float) – the learning rate
- **momentum** (float) – optimizer momentum

- **weight_decay** (float) – optimizer weight decay
- **datamodule** – the DataModule (train, val, test dataloaders)
- **data_dir** (str) – the directory to store data
- **batch_size** (int) – batch size
- **use_mlp** (bool) – add an mlp to the encoders
- **num_workers** (int) – workers for the loaders

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the

`lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
```

(continues on next page)

(continued from previous page)

```
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

---

forward(*img_q*, *img_k*, *queue*)

Input: im_q: a batch of query images im_k: a batch of key images queue: a queue from which to pick negative samples

Output: logits, targets

init_encoders(*base_encoder*)
Override to add your own encoders.

training_step(*batch*, *batch_idx*)
Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (`int`) – Integer displaying index of this batch
- **optimizer_idx** (`int`) – When using multiple optimizers, this argument will also be present.
- **hiddens** (`Any`) – Passed in if `truncated_bptt_steps` > 0.

Returns

Any of.

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- **`None` - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
```

(continues on next page)

(continued from previous page)

```
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

---

validation_epoch_end(*outputs*)

Called at the end of the validation epoch with the outputs of all validation steps.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters **outputs** – List of outputs you defined in `validation_step()`, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader.

Returns None

Note: If you didn't define a `validation_step()`, this won't be called.

Examples

With a single dataloader:

```
def validation_epoch_end(self, val_step_outputs):
    for out in val_step_outputs:
        ...
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each validation step for that dataloader.

```
def validation_epoch_end(self, outputs):
    for dataloader_output_result in outputs:
        dataloader_outs = dataloader_output_result.dataloader_i_outputs

    self.log("final_metric", final_value)
```

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Returns

- Any object or value
- `None` - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...
```

(continues on next page)

(continued from previous page)

```
# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, `validation_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

20.2.6 SimCLR

PyTorch Lightning implementation of SimCLR

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

Model implemented by:

- William Falcon
- Tullie Murrell
- Ananya Harsh Jha

To Train:

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

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

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

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

CIFAR-10 baseline

Table 2: Cifar-10 implementation results

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

CIFAR-10 pretrained model:

```
from pl_bolts.models.self_supervised import SimCLR

weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/simclr/simclr-cifar10-
↪sgd/simclr-cifar10-sgd.ckpt'
```

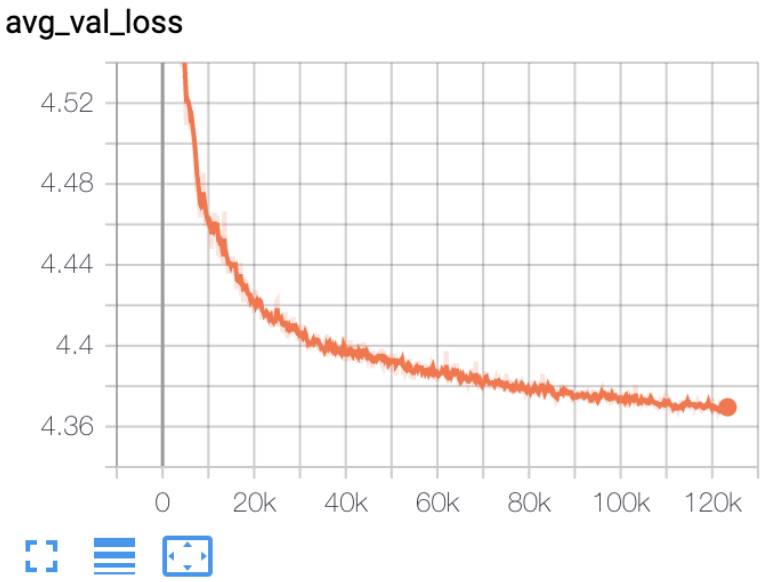
(continues on next page)

(continued from previous page)

```
simclr = SimCLR.load_from_checkpoint(weight_path, strict=False)

simclr.freeze()
```

Pre-training:

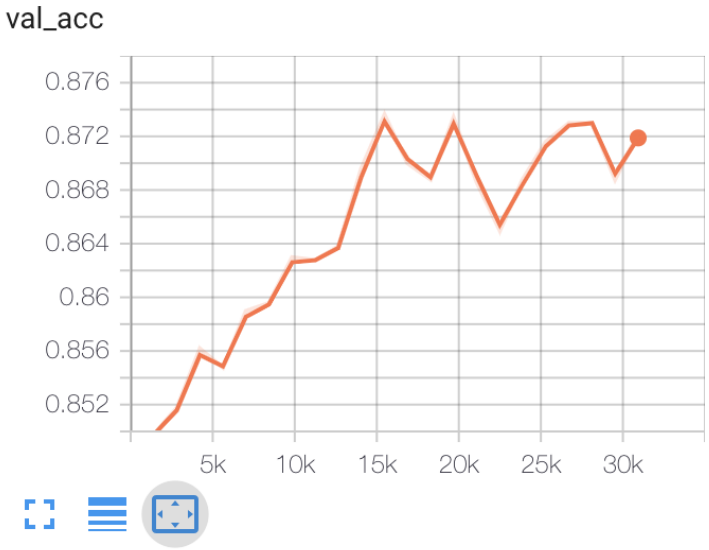


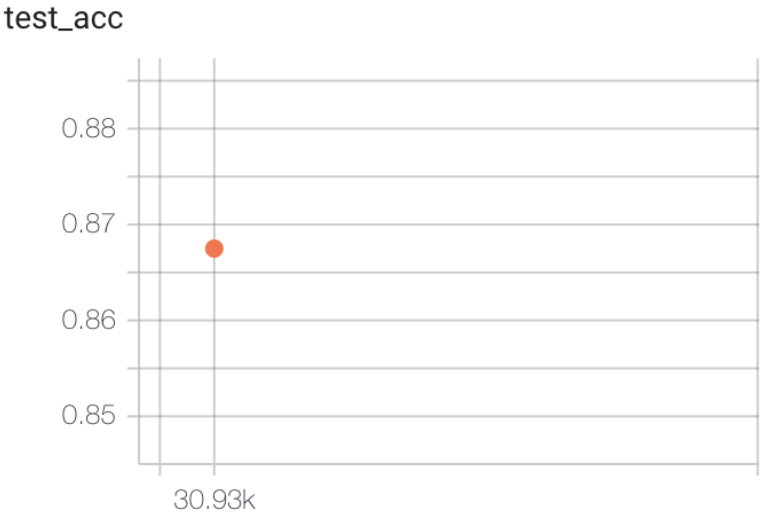
Fine-tuning (Single layer MLP, 1024 hidden units):

To reproduce:

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

(continues on next page)


val_acc
0.876
0.872
0.868
0.864
0.86
0.856
0.852
5k
10k
15k
20k
25k
30k


test_acc
0.88
0.87
0.86
0.85
30.93k

(continued from previous page)

```
    --online_ft

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

Imagenet baseline for SimCLR

Table 3: Cifar-10 implementation results

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

Imagenet pretrained model:

```
from pl_bolts.models.self_supervised import SimCLR

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

simclr.freeze()
```

To reproduce:

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

# finetune
python simclr_finetuner.py
    --gpus 8
```

(continues on next page)

(continued from previous page)

```
    --ckpt_path path/to/simclr/ckpt
    --dataset imagenet
    --data_dir path/to/imagenet/dataset
    --batch_size 256
    --num_workers 16
    --learning_rate 0.8
    --nesterov True
    --num_epochs 90
```

SimCLR API

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

Bases: pytorch_lightning.core.module.LightningModule

> **Warning:** The feature SimCLR is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Parameters

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

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple lr_scheduler_config).
- **Dictionary**, with an "optimizer" key, and (optionally) a "lr_scheduler" key whose value is a single LR scheduler or lr_scheduler_config.
- **Tuple of dictionaries** as described above, with an optional "frequency" key.

- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
```

(continues on next page)

(continued from previous page)

```
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
```

(continues on next page)

(continued from previous page)

```
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

---

forward(*x*)

Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns Your model's output

nt_xent_loss(*out_1*, *out_2*, *temperature*, *eps=1e-06*)
assume out_1 and out_2 are normalized out_1: [batch_size, dim] out_2: [batch_size, dim]

training_step(*batch*, *batch_idx*)
Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (Tensor | (Tensor, ...) | [Tensor, ...]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (int) – Integer displaying index of this batch
- **optimizer_idx** (int) – When using multiple optimizers, this argument will also be present.
- **hiddens** (Any) – Passed in if truncated_bptt_steps > 0.

Returns

Any of.

- Tensor - The loss tensor
- dict - A dictionary. Can include any keys, but must include the key 'loss'
- **None - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional optimizer_idx parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
```

(continues on next page)

(continued from previous page)

```
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Returns

- Any object or value
- `None` - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...


# if you have multiple val dataloaders:
```

(continues on next page)

(continued from previous page)

```
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, `validation_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

20.2.7 SwAV

PyTorch Lightning implementation of SwAV Adapted from the official implementation

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

Implementation adapted by:

- Ananya Harsh Jha

To Train:

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

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

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

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

# model
model = SwAV(
    gpus=1,
    num_samples=dm.num_unlabeled_samples,
    dataset='stl10',
    batch_size=batch_size,
    num_crops=(2,4)
)

# fit
trainer = pl.Trainer(precision=16, accelerator='auto')
trainer.fit(model, datamodule=dm)
```

Pre-trained ImageNet

We have included an option to directly load ImageNet weights provided by FAIR into bolts.

You can load the pretrained model using:

ImageNet pretrained model:

```
from pl_bolts.models.self_supervised import SwAV

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

swav.freeze()
```

STL-10 baseline

The original paper does not provide baselines on STL10.

Table 4: STL-10 implementation results

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

- Pre-training tensorboard link

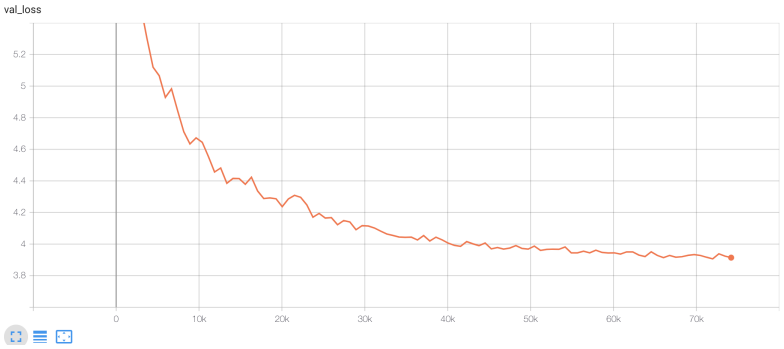
STL-10 pretrained model:

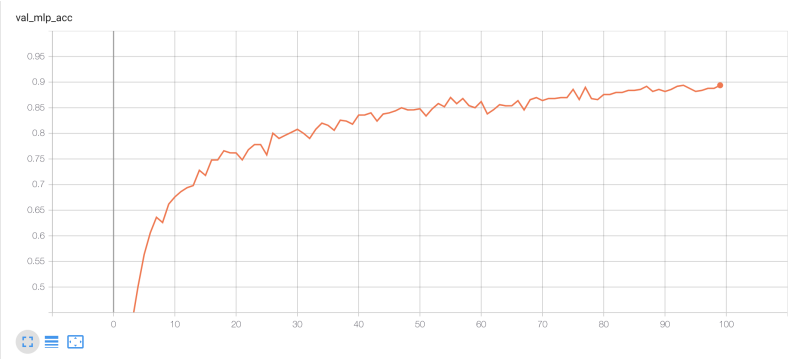
```
from pl_bolts.models.self_supervised import SwAV

weight_path = 'https://pl-bolts-weights.s3.us-east-2.amazonaws.com/swav/checkpoints/swav_
↪stl10.pth.tar'
swav = SwAV.load_from_checkpoint(weight_path, strict=False)

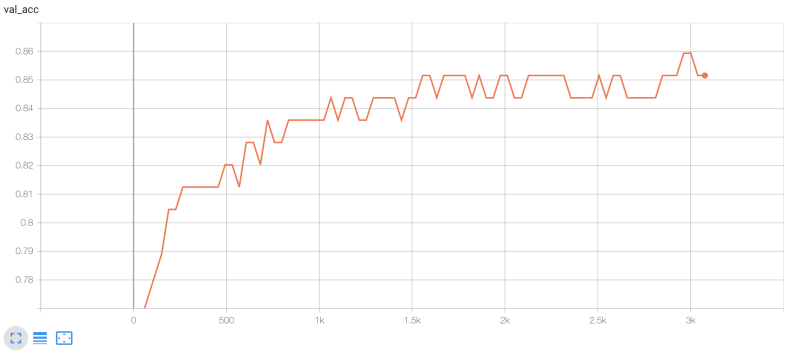
swav.freeze()
```

Pre-training:





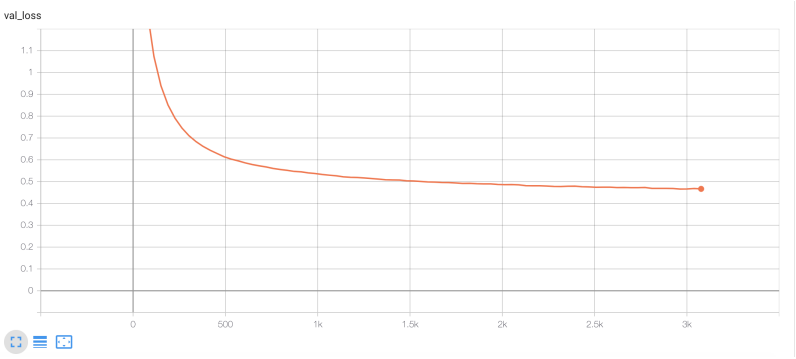
Fine-tuning (Single layer MLP, 1024 hidden units):



To reproduce:

```
# pretrain
python swav_module.py
    --online_ft
    --gpus 1
    --batch_size 128
    --learning_rate 1e-3
    --gaussian_blur
    --queue_length 0
```

(continues on next page)



(continued from previous page)

```
    --jitter_strength 1.
    --num_prototypes 512

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

Imagenet baseline for SwAV

Table 5: Cifar-10 implementation results

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

Imagenet pretrained model:

```
from pl_bolts.models.self_supervised import SwAV

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

swav.freeze()
```

SwAV API

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

Bases: pytorch_lightning.core.module.LightningModule

Parameters

- **gpus** (int) – number of gpus per node used in training, passed to SwAV module to manage the queue and select distributed sinkhorn
- **num_nodes** (int) – number of nodes to train on
- **num_samples** (int) – number of image samples used for training
- **batch_size** (int) – batch size per GPU in ddp
- **dataset** (str) – dataset being used for train/val
- **arch** (str) – encoder architecture used for pre-training
- **hidden_mlp** (int) – hidden layer of non-linear projection head, set to 0 to use a linear projection head
- **feat_dim** (int) – output dim of the projection head
- **warmup_epochs** (int) – apply linear warmup for this many epochs
- **max_epochs** (int) – epoch count for pre-training
- **num_prototypes** (int) – count of prototype vectors
- **freeze_prototypes_epochs** (int) – epoch till which gradients of prototype layer are frozen
- **temperature** (float) – loss temperature
- **sinkhorn_iterations** (int) – iterations for sinkhorn normalization
- **queue_length** (int) – set queue when batch size is small, must be divisible by total batch-size (i.e. total_gpus * batch_size), set to 0 to remove the queue
- **queue_path** (str) – folder within the logs directory
- **epoch_queue_starts** (int) – start uing the queue after this epoch
- **crops_for_assign** (tuple) – list of crop ids for computing assignment
- **num_crops** (tuple) – number of global and local crops, ex: [2, 6]
- **first_conv** (bool) – keep first conv same as the original resnet architecture, if set to false it is replace by a kernel 3, stride 1 conv (cifar-10)

- **maxpool1** (bool) – keep first maxpool layer same as the original resnet architecture, if set to false, first maxpool is turned off (cifar10, maybe stl10)
- **optimizer** (str) – optimizer to use
- **exclude_bn_bias** (bool) – exclude batchnorm and bias layers from weight decay in optimizers
- **start_lr** (float) – starting lr for linear warmup
- **learning_rate** (float) – learning rate
- **final_lr** (float) – float = final learning rate for cosine weight decay
- **weight_decay** (float) – weight decay for optimizer
- **epsilon** (float) – epsilon val for swav assignments

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
```

(continues on next page)

(continued from previous page)

```
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
```

(continues on next page)

(continued from previous page)

```
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

forward(*x*)

Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Returns Your model's output

on_after_backward()

Called after `loss.backward()` and before optimizers are stepped.

Note: If using native AMP, the gradients will not be unscaled at this point. Use the `on_before_optimizer_step` if you need the unscaled gradients.

on_train_epoch_end()

Called in the training loop at the very end of the epoch.

To access all batch outputs at the end of the epoch, either:

1. Implement *training_epoch_end* in the LightningModule OR
2. Cache data across steps on the attribute(s) of the *LightningModule* and access them in this hook

Return type `None`

on_train_epoch_start()

Called in the training loop at the very beginning of the epoch.

setup(*stage*)

Called at the beginning of fit (train + validate), validate, test, or predict. This is a good hook when you need to build models dynamically or adjust something about them. This hook is called on every process when using DDP.

Parameters **stage** – either 'fit', 'validate', 'test', or 'predict'

Example:

```
class LitModel(...):
    def __init__(self):
        self.l1 = None

    def prepare_data(self):
        download_data()
        tokenize()

        # don't do this
        self.something = else

    def setup(self, stage):
        data = load_data(...)
        self.l1 = nn.Linear(28, data.num_classes)
```

training_step(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (Tensor | (Tensor, ...) | [Tensor, ...]) – The output of your DataLoader. A tensor, tuple or list.
- **batch_idx** (int) – Integer displaying index of this batch
- **optimizer_idx** (int) – When using multiple optimizers, this argument will also be present.
- **hiddens** (Any) – Passed in if truncated_bptt_steps > 0.

Returns

Any of.

- Tensor - The loss tensor
- dict - A dictionary. Can include any keys, but must include the key 'loss'
- **None - Training will skip to the next batch. This is only for automatic optimization.**
  This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional optimizer_idx parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

---

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** – The output of your `DataLoader`.
- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Returns

- Any object or value
- `None` - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, `validation_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

20.2.8 SimSiam

class pl_bolts.models.self_supervised.**SimSiam**(*learning_rate=0.05*, *weight_decay=0.0001*, *momentum=0.9*, *warmup_epochs=10*, *max_epochs=100*, *base_encoder='resnet50'*, *encoder_out_dim=2048*, *projector_hidden_dim=2048*, *projector_out_dim=2048*, *predictor_hidden_dim=512*, *exclude_bn_bias=False*, ***kwargs*)

Bases: `pytorch_lightning.core.module.LightningModule`

PyTorch Lightning implementation of Exploring Simple Siamese Representation Learning (SimSiam)_

Paper authors: Xinlei Chen, Kaiming He.

Parameters

- **learning_rate** (*float, optional*) – optimizer leaning rate. Defaults to 0.05.
- **weight_decay** (*float, optional*) – optimizer weight decay. Defaults to 1e-4.
- **momentum** (*float, optional*) – optimizer momentum. Defaults to 0.9.
- **warmup_epochs** (*int, optional*) – number of epochs for scheduler warmup. Defaults to 10.
- **max_epochs** (*int, optional*) – maximum number of epochs for scheduler. Defaults to 100.
- **base_encoder** (*Union[str, nn.Module], optional*) – base encoder architecture. Defaults to "resnet50".
- **encoder_out_dim** (*int, optional*) – base encoder output dimension. Defaults to 2048.
- **projector_hidden_dim** (*int, optional*) – projector MLP hidden dimension. Defaults to 2048.
- **projector_out_dim** (*int, optional*) – project MLP output dimension. Defaults to 2048.
- **predictor_hidden_dim** (*int, optional*) – predictor MLP hidden dimension. Defaults to 512.
- **exclude_bn_bias** (*bool, optional*) – option to exclude batchnorm and bias terms from weight decay. Defaults to False.

Model implemented by:

- Zvi Lapp

Example:

```
model = SimSiam()

dm = CIFAR10DataModule(num_workers=0)
```

(continues on next page)

(continued from previous page)

```
dm.train_transforms = SimCLRTrainDataTransform(32)
dm.val_transforms = SimCLREvalDataTransform(32)

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

CLI command:

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

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

calculate_loss(*v_online*, *v_target*)
Calculates similarity loss between the online network prediction of target network projection.

Parameters

- **v_online** (*Tensor*) – Online network view
- **v_target** (*Tensor*) – Target network view

Return type Tensor

configure_optimizers()
Configure optimizer and learning rate scheduler.

static exclude_from_weight_decay(*named_params*, *weight_decay*, *skip_list=('bias', 'bn')*)
Exclude parameters from weight decay.

Return type List[Dict]

forward(*x*)
Returns encoded representation of a view.

Return type Tensor

training_step(*batch*, *batch_idx*)
Complete training loop.

Return type Tensor

validation_step(*batch*, *batch_idx*)
Complete validation loop.

Return type Tensor

CHAPTER

TWENTYONE

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.

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

21.1 Linear Regression

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

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

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

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

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

model = LinearRegression(input_dim=10)
trainer = pl.Trainer()
trainer.fit(model, train_dataloaders=loaders.train_dataloader(), val_dataloaders=loaders.
↪val_dataloader())
trainer.test(test_dataloaders=loaders.test_dataloader())
```

class pl_bolts.models.regression.linear_regression.**LinearRegression**(*input_dim*, *output_dim=1*, *bias=True*, *learning_rate=0.0001*, *optimizer=<class 'torch.optim.adam.Adam'>*, *l1_strength=0.0*, *l2_strength=0.0*, ***kwargs*)

Bases: pytorch_lightning.core.module.LightningModule

> **Warning:** The feature LinearRegression is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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

Parameters

- **input_dim** (int) – number of dimensions of the input (1+)
- **output_dim** (int) – number of dimensions of the output (default: 1)
- **bias** (bool) – If false, will not use $+b$
- **learning_rate** (float) – learning_rate for the optimizer
- **optimizer** (Type[Optimizer]) – the optimizer to use (default: Adam)
- **l1_strength** (float) – L1 regularization strength (default: 0.0)
- **l2_strength** (float) – L2 regularization strength (default: 0.0)

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Return type Optimizer

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple lr_scheduler_config).
- **Dictionary**, with an "optimizer" key, and (optionally) a "lr_scheduler" key whose value is a single LR scheduler or lr_scheduler_config.
- **Tuple of dictionaries** as described above, with an optional "frequency" key.
- **None** - Fit will run without any optimizer.

The lr_scheduler_config is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
```

(continues on next page)

(continued from previous page)

```
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]
```

(continues on next page)

(continued from previous page)

```
# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

forward(*x*)

Same as `torch.nn.Module.forward()`.

Parameters

- ***args** – Whatever you decide to pass into the forward method.
- ****kwargs** – Keyword arguments are also possible.

Return type `Tensor`

Returns Your model's output

test_epoch_end(*outputs*)

Called at the end of a test epoch with the output of all test steps.

```
# the pseudocode for these calls
test_outs = []
for test_batch in test_data:
    out = test_step(test_batch)
    test_outs.append(out)
test_epoch_end(test_outs)
```

Parameters **outputs** (List[Dict[str, Tensor]]) – List of outputs you defined in `test_step_end()`, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader

Return type Dict[str, Tensor]

Returns None

Note: If you didn't define a `test_step()`, this won't be called.

Examples

With a single dataloader:

```
def test_epoch_end(self, outputs):
    # do something with the outputs of all test batches
    all_test_preds = test_step_outputs.predictions

    some_result = calc_all_results(all_test_preds)
    self.log(some_result)
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each test step for that dataloader.

```
def test_epoch_end(self, outputs):
    final_value = 0
    for dataloader_outputs in outputs:
        for test_step_out in dataloader_outputs:
            # do something
            final_value += test_step_out

    self.log("final_metric", final_value)
```

test_step(*batch*, *batch_idx*)

Operates on a single batch of data from the test set. In this step you'd normally generate examples or calculate anything of interest such as accuracy.

```
# the pseudocode for these calls
test_outs = []
for test_batch in test_data:
    out = test_step(test_batch)
```

(continues on next page)

(continued from previous page)

```
    test_outs.append(out)
test_epoch_end(test_outs)
```

Parameters

- **batch** (Tuple[Tensor, Tensor]) – The output of your DataLoader.
- **batch_idx** (int) – The index of this batch.
- **dataloader_id** – The index of the dataloader that produced this batch. (only if multiple test dataloaders used).

Return type Dict[str, Tensor]

Returns

Any of.

- Any object or value
- None - Testing will skip to the next batch

```
# if you have one test dataloader:
def test_step(self, batch, batch_idx):
    ...


# if you have multiple test dataloaders:
def test_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single test dataset
def test_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    test_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'test_loss': loss, 'test_acc': test_acc})
```

If you pass in multiple test dataloaders, `test_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple test dataloaders
def test_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to test you don't need to implement this method.

Note: When the `test_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of the test epoch, the model goes back to training mode and gradients are enabled.

`training_step`(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (`int`) – Integer displaying index of this batch
- **optimizer_idx** (`int`) – When using multiple optimizers, this argument will also be present.
- **hiddens** (`Any`) – Passed in if `truncated_bptt_steps` > 0.

Return type `Dict[str, Tensor]`

Returns

Any of.

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- **`None` - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
```

(continues on next page)

(continued from previous page)

```
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

---

validation_epoch_end(*outputs*)

Called at the end of the validation epoch with the outputs of all validation steps.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters **outputs** (`List[Dict[str, Tensor]]`) – List of outputs you defined in `validation_step()`, or if there are multiple dataloaders, a list containing a list of outputs for each dataloader.

Return type `Dict[str, Tensor]`

Returns None

Note: If you didn't define a `validation_step()`, this won't be called.

Examples

With a single dataloader:

```
def validation_epoch_end(self, val_step_outputs):
    for out in val_step_outputs:
        ...
```

With multiple dataloaders, *outputs* will be a list of lists. The outer list contains one entry per dataloader, while the inner list contains the individual outputs of each validation step for that dataloader.

```
def validation_epoch_end(self, outputs):
    for dataloader_output_result in outputs:
        dataloader_outs = dataloader_output_result.dataloader_i_outputs

    self.log("final_metric", final_value)
```

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** (Tuple[Tensor, Tensor]) – The output of your DataLoader.
- **batch_idx** (int) – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Return type Dict[str, Tensor]

Returns

- Any object or value
- None - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
```

(continues on next page)

(continued from previous page)

```
    ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, `validation_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

21.2 Logistic Regression

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

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

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

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

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

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

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

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

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

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

model = LogisticRegression(input_dim=28 * 28, num_classes=10, learning_rate=0.001)
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=<class 'torch.optim.adam.Adam'>*, *l1_strength=0.0*, *l2_strength=0.0*, ***kwargs*)

Bases: pytorch_lightning.core.module.LightningModule

Logistic Regression Model.

Logistic Regression.

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** (Type[Optimizer]) – Model optimizer to use.
- **l1_strength** (float) – L1 regularization strength.
- **l2_strength** (float) – L2 regularization strength.

linear
Linear layer.

Type torch.nn.modules.linear.Linear

criterion
Cross-Entropy loss function.

Type torch.nn.modules.loss.CrossEntropyLoss

optimizer
Model optimizer to use.

static add_model_specific_args(*parent_parser*)
Adds model specific arguments to the parser.

Parameters **parent_parser** (ArgumentParser) – Parent parser to which the arguments will be added.

Return type ArgumentParser

Returns ArgumentParser with the added arguments.

configure_optimizers()
Configure the optimizer for the model.

Return type Optimizer

Returns Optimizer.

forward(*x*)
Forward pass of the model.

Parameters **x** (Tensor) – Input tensor.

Return type Tensor

Returns Output tensor.

test_epoch_end(*outputs*)
Test epoch end for the model.

Parameters **outputs** (List[Dict[str, Tensor]]) – List of outputs from the test step.

Return type Dict[str, Tensor]

Returns Loss tensor.

test_step(*batch*, *batch_idx*)
Test step for the model.

Parameters

- **batch** (Tuple[Tensor, Tensor]) – Batch of data.
- **batch_idx** (int) – Batch index.

Return type Dict[str, Tensor]

Returns Loss tensor.

training_step(*batch*, *batch_idx*)
Training step for the model.

Parameters

- **batch** (Tuple[Tensor, Tensor]) – Batch of data.
- **batch_idx** (int) – Batch index.

Return type Dict[str, Tensor]

Returns Loss tensor.

validation_epoch_end(*outputs*)
Validation epoch end for the model.

Parameters **outputs** (List[Dict[str, Tensor]]) – List of outputs from the validation step.

Return type Dict[str, Tensor]

Returns Loss tensor.

validation_step(*batch*, *batch_idx*)
Validation step for the model.

Parameters

- **batch** (Tuple[Tensor, Tensor]) – Batch of data.
- **batch_idx** (int) – Batch index.

Return type Dict[str, Tensor]

Returns Loss tensor.

CHAPTER

TWENTYTWO

LINEAR WARMUP COSINE ANNEALING

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

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

Warning: The feature LinearWarmupCosineAnnealingLR is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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

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

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

Example

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

Parameters

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

`get_lr()`

Compute learning rate using chainable form of the scheduler.

Return type `List[float]`

CHAPTER

TWENTYTHREE

SELF-SUPERVISED LEARNING

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

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

23.1 CPC transforms

Transforms used for CPC

23.1.1 CIFAR-10 Train (c)

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

Bases: `object`

Warning: The feature CPCTrainTransformsCIFAR10 is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Transforms used for CPC:

Transforms:

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

Example:

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

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

Parameters

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

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

Return type Tensor

23.1.2 CIFAR-10 Eval (c)

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

Bases: object

Warning: The feature CPCEvalTransformsCIFAR10 is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Transforms used for CPC:

Transforms:

```
random_flip
transforms.ToTensor()
normalize
Patchify(patch_size=patch_size, overlap_size=overlap)
```

Example:

```
# in a regular dataset
CIFAR10(..., transforms=CPCEvalTransformsCIFAR10())

# in a DataModule
module = CIFAR10DataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,␣
↪transforms=CPCEvalTransformsCIFAR10())
```

Parameters

- **patch_size** (int) – size of patches when cutting up the image into overlapping patches

- **overlap** (int) – how much to overlap patches

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

Return type Tensor

23.1.3 Imagenet Train (c)

class pl_bolts.transforms.self_supervised.cpc_transforms.**CPCTrainTransformsImageNet128**(*patch_size=32*, *overlap=16*)

Bases: object

> **Warning:** The feature CPCTrainTransformsImageNet128 is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Transforms used for CPC:

Transforms:

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

Example:

```
# in a regular dataset
Imagenet(..., transforms=CPCTrainTransformsImageNet128())

# in a DataModule
module = ImagenetDataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,␣
↪transforms=CPCTrainTransformsImageNet128())
```

Parameters

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

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

Return type Tensor

23.1.4 Imagenet Eval (c)

class pl_bolts.transforms.self_supervised.cpc_transforms.**CPCEvalTransformsImageNet128**(*patch_size=32*, *overlap=16*)

Bases: object

> **Warning:** The feature CPCEvalTransformsImageNet128 is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Transforms used for CPC:

Transforms:

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

Example:

```
# in a regular dataset
Imagenet(..., transforms=CPCEvalTransformsImageNet128())

# in a DataModule
module = ImagenetDataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,␣
↪transforms=CPCEvalTransformsImageNet128())
```

Parameters

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

__call__(*inp*)

Call self as a function.

Return type Tensor

23.1.5 STL-10 Train (c)

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

Bases: object

> **Warning:** The feature CPCTrainTransformsSTL10 is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Transforms used for CPC:

Transforms:

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

Example:

```
# in a regular dataset
STL10(..., transforms=CPCTrainTransformsSTL10())

# in a DataModule
module = STL10DataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,␣
↪transforms=CPCTrainTransformsSTL10())
```

Parameters

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

__call__(*inp*)

Call self as a function.

Return type Tensor

23.1.6 STL-10 Eval (c)

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

Bases: object

Warning: The feature CPCEvalTransformsSTL10 is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Transforms used for CPC:

Transforms:

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

Example:

```
# in a regular dataset
STL10(..., transforms=CPCEvalTransformsSTL10())

# in a DataModule
module = STL10DataModule(PATH)
train_loader = module.train_dataloader(batch_size=32,␣
↪transforms=CPCEvalTransformsSTL10())
```

Parameters

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

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

Return type Tensor

23.2 AMDIM transforms

Transforms used for AMDIM

23.2.1 CIFAR-10 Train (a)

class pl_bolts.transforms.self_supervised.amdim_transforms.**AMDIMTrainTransformsCIFAR10**
Bases: object

Warning: The feature AMDIMTrainTransformsCIFAR10 is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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.

Return type Tuple[Tensor, Tensor]

23.2.2 CIFAR-10 Eval (a)

class pl_bolts.transforms.self_supervised.amdim_transforms.**AMDIMEvalTransformsCIFAR10**
Bases: object

> **Warning:** The feature AMDIMEvalTransformsCIFAR10 is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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.

Return type Tensor

23.2.3 Imagenet Train (a)

class pl_bolts.transforms.self_supervised.amdim_transforms.**AMDIMTrainTransformsImageNet128**(*height=128*)
Bases: object

> **Warning:** The feature AMDIMTrainTransformsImageNet128 is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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.

Return type Tuple[Tensor, Tensor]

23.2.4 Imagenet Eval (a)

class pl_bolts.transforms.self_supervised.amdim_transforms.**AMDIMEvalTransformsImageNet128**(*height=128*)
Bases: object

Warning: The feature AMDIMEvalTransformsImageNet128 is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Transforms applied to AMDIM.

Transforms:

```
transforms.Resize(height + 6, interpolation=InterpolationMode.BICUBIC),
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.

Return type Tensor

23.2.5 STL-10 Train (a)

class pl_bolts.transforms.self_supervised.amdim_transforms.**AMDIMTrainTransformsSTL10**(*height=64*)
Bases: object

Warning: The feature AMDIMTrainTransformsSTL10 is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality

may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

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.

Return type Tuple[Tensor, Tensor]

23.2.6 STL-10 Eval (a)

class pl_bolts.transforms.self_supervised.amdim_transforms.**AMDIMEvalTransformsSTL10**(*height=64*)
Bases: object

Warning: The feature AMDIMEvalTransformsSTL10 is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Transforms applied to AMDIM.

Transforms:

```
transforms.Resize(height + 6, interpolation=InterpolationMode.BICUBIC),
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.

Return type Tensor

23.3 MOCO V2 transforms

Transforms used for MOCO V2

23.3.1 CIFAR-10 Train (m2)

class pl_bolts.transforms.self_supervised.moco_transforms.**MoCo2TrainCIFAR10Transforms**(*size=32*)

Bases: object

> **Warning:** The feature MoCo2TrainCIFAR10Transforms is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

MoCo v2 transforms.

Parameters **size** (*int, optional*) – input size. Defaults to 32.

Transform:

```
RandomResizedCrop(size=self.input_size)
```

Example:

```
from pl_bolts.transforms.self_supervised.MoCo_transforms import␣
↪MoCo2TrainCIFAR10Transforms

transform = MoCo2TrainCIFAR10Transforms(input_size=32)
x = sample()
(xi, xj) = transform(x)
```

MoCo 2 augmentation:

https://arxiv.org/pdf/2003.04297.pdf

__call__(*x*)

Call self as a function.

Return type Tuple[Tensor, Tensor]

23.3.2 CIFAR-10 Eval (m2)

class pl_bolts.transforms.self_supervised.moco_transforms.**MoCo2EvalCIFAR10Transforms**(*size=32*)

Bases: object

> **Warning:** The feature MoCo2EvalCIFAR10Transforms is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

MoCo 2 augmentation:

https://arxiv.org/pdf/2003.04297.pdf

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

Return type Tuple[Tensor, Tensor]

23.3.3 Imagenet Train (m2)

class pl_bolts.transforms.self_supervised.moco_transforms.**MoCo2TrainSTL10Transforms**(*size=64*)
Bases: object

> **Warning:** The feature MoCo2TrainSTL10Transforms is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

MoCo 2 augmentation:

https://arxiv.org/pdf/2003.04297.pdf

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

Return type Tuple[Tensor, Tensor]

23.3.4 Imagenet Eval (m2)

class pl_bolts.transforms.self_supervised.moco_transforms.**MoCo2EvalSTL10Transforms**(*size=64*)
Bases: object

> **Warning:** The feature MoCo2EvalSTL10Transforms is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

MoCo 2 augmentation:

https://arxiv.org/pdf/2003.04297.pdf

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

Return type Tuple[Tensor, Tensor]

23.3.5 STL-10 Train (m2)

class pl_bolts.transforms.self_supervised.moco_transforms.**MoCo2TrainImagenetTransforms**(*size=224*)
Bases: object

> **Warning:** The feature MoCo2TrainImagenetTransforms is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

MoCo 2 augmentation:

https://arxiv.org/pdf/2003.04297.pdf

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

Return type Tuple[Tensor, Tensor]

23.3.6 STL-10 Eval (m2)

class pl_bolts.transforms.self_supervised.moco_transforms.**MoCo2EvalImagenetTransforms**(*size=128*)
Bases: object

> **Warning:** The feature MoCo2EvalImagenetTransforms is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Transforms for MoCo during training step.

https://arxiv.org/pdf/2003.04297.pdf

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

Return type Tuple[Tensor, Tensor]

23.4 SimCLR transforms

Transforms used for SimCLR

23.4.1 Train (sc)

class pl_bolts.transforms.self_supervised.simclr_transforms.**SimCLRTrainDataTransform**(*input_height=224*, *gaussian_blur=True*, *jitter_strength=1.0*, *normalize=None*)

Bases: object

Transforms for SimCLR during training step of the pre-training stage.

Parameters

- **input_height** (*int, optional*) – expected output size of image. Defaults to 224.
- **gaussian_blur** (*bool, optional*) – applies Gaussian blur if True. Defaults to True.
- **jitter_strength** (*float, optional*) – color jitter multiplier. Defaults to 1.0.
- **normalize** (*Callable, optional*) – optional transform to normalize. Defaults to None.

Transform:

```
RandomResizedCrop(size=self.input_height)
RandomHorizontalFlip()
RandomApply([color_jitter], p=0.8)
RandomGrayscale(p=0.2)
RandomApply([GaussianBlur(kernel_size=int(0.1 * self.input_height))], p=0.5)
transforms.ToTensor()
```

Example:

```
from pl_bolts.transforms.self_supervised.simclr_transforms import␣
↪SimCLRTrainDataTransform

transform = SimCLRTrainDataTransform(input_height=32)
x = sample()
(xi, xj, xk) = transform(x) # xk is only for the online evaluator if used
```

__call__(*sample*)

Call self as a function.

Return type Tuple[Tensor, Tensor, Tensor]

23.4.2 Eval (sc)

class pl_bolts.transforms.self_supervised.simclr_transforms.**SimCLREvalDataTransform**(*input_height=224*, *gaussian_blur=True*, *jitter_strength=1.0*, *normalize=None*)

Bases: pl_bolts.transforms.self_supervised.simclr_transforms.SimCLRTrainDataTransform

Transforms for SimCLR during the validation step of the pre-training stage.

Parameters

- **input_height** (*int, optional*) – expected output size of image. Defaults to 224.
- **gaussian_blur** (*bool, optional*) – applies Gaussian blur if True. Defaults to True.
- **jitter_strength** (*float, optional*) – color jitter multiplier. Defaults to 1.0.
- **normalize** (*Callable, optional*) – optional transform to normalize. Defaults to None.

Transform:

```
Resize(input_height + 10, interpolation=3)
transforms.CenterCrop(input_height),
transforms.ToTensor()
```

Example:

```
from pl_bolts.transforms.self_supervised.simclr_transforms import␣
↪SimCLREvalDataTransform

transform = SimCLREvalDataTransform(input_height=32)
x = sample()
(xi, xj, xk) = transform(x) # xk is only for the online evaluator if used
```

23.5 Identity class

Example:

```
from pl_bolts.utils import Identity
```

class pl_bolts.utils.self_supervised.**Identity**

Bases: torch.nn.modules.module.Module

> **Warning:** The feature Identity is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

An identity class to replace arbitrary layers in pretrained models.

Example:

```
from pl_bolts.utils import Identity

model = resnet18()
model.fc = Identity()
```

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type `Tensor`

23.6 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*)

Warning: The feature torchvision_ssl_encoder is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Return type `Module`

23.7 SSL backbone finetuner

class pl_bolts.models.self_supervised.ssl_finetuner.**SSLFineTuner**(*backbone*, *in_features=2048*, *num_classes=1000*, *epochs=100*, *hidden_dim=None*, *dropout=0.0*, *learning_rate=0.1*, *weight_decay=1e-06*, *nesterov=False*, *scheduler_type='cosine'*, *decay_epochs=(60, 80)*, *gamma=0.1*, *final_lr=0.0*)

Bases: pytorch_lightning.core.module.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 CPC_v2
from pl_bolts.datamodules import CIFAR10DataModule
from pl_bolts.models.self_supervised.cpc.transforms import CPCEvalTransformsCIFAR10,
↪CPCTrainTransformsCIFAR10

# pretrained model
backbone = CPC_v2.load_from_checkpoint(PATH, strict=False)

# dataset + transforms
dm = CIFAR10DataModule(data_dir='.')
dm.train_transforms = CPCTrainTransformsCIFAR10()
dm.val_transforms = CPCEvalTransformsCIFAR10()

# finetuner
finetuner = SSLFineTuner(backbone, in_features=backbone.z_dim, num_classes=backbone.
↪num_classes)

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

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

Parameters

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

`configure_optimizers()`

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
```

(continues on next page)

(continued from previous page)

```
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer as needed.
- If learning rate scheduler is specified in `configure_optimizers()` with key `"interval"` (default "epoch") in the scheduler configuration, Lightning will call the scheduler's `.step()` method automatically in case of automatic optimization.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.

- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

---

on_train_epoch_start()
Called in the training loop at the very beginning of the epoch.

Return type None

test_step(*batch*, *batch_idx*)
Operates on a single batch of data from the test set. In this step you'd normally generate examples or calculate anything of interest such as accuracy.

```
# the pseudocode for these calls
test_outs = []
for test_batch in test_data:
    out = test_step(test_batch)
    test_outs.append(out)
test_epoch_end(test_outs)
```

Parameters

- **batch** – The output of your DataLoader.
- **batch_idx** – The index of this batch.
- **dataloader_id** – The index of the dataloader that produced this batch. (only if multiple test dataloaders used).

Returns

Any of.

- Any object or value
- None - Testing will skip to the next batch

```
# if you have one test dataloader:
def test_step(self, batch, batch_idx):
    ...


# if you have multiple test dataloaders:
def test_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single test dataset
def test_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)
```

(continues on next page)

(continued from previous page)

```
    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    test_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'test_loss': loss, 'test_acc': test_acc})
```

If you pass in multiple test dataloaders, `test_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple test dataloaders
def test_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to test you don't need to implement this method.

Note: When the `test_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of the test epoch, the model goes back to training mode and gradients are enabled.

`training_step`(*batch*, *batch_idx*)

Here you compute and return the training loss and some additional metrics for e.g. the progress bar or logger.

Parameters

- **batch** (`Tensor` | (`Tensor`, ...) | [`Tensor`, ...]) – The output of your `DataLoader`. A tensor, tuple or list.
- **batch_idx** (`int`) – Integer displaying index of this batch
- **optimizer_idx** (`int`) – When using multiple optimizers, this argument will also be present.
- **hiddens** (`Any`) – Passed in if `truncated_bptt_steps` > 0.

Returns

Any of.

- `Tensor` - The loss tensor
- `dict` - A dictionary. Can include any keys, but must include the key `'loss'`
- **`None` - Training will skip to the next batch. This is only for automatic optimization.** This is not supported for multi-GPU, TPU, IPU, or DeepSpeed.

In this step you'd normally do the forward pass and calculate the loss for a batch. You can also do fancier things like multiple forward passes or something model specific.

Example:

```
def training_step(self, batch, batch_idx):
    x, y, z = batch
    out = self.encoder(x)
    loss = self.loss(out, x)
    return loss
```

If you define multiple optimizers, this step will be called with an additional `optimizer_idx` parameter.

```
# Multiple optimizers (e.g.: GANs)
def training_step(self, batch, batch_idx, optimizer_idx):
    if optimizer_idx == 0:
        # do training_step with encoder
        ...
    if optimizer_idx == 1:
        # do training_step with decoder
        ...
```

If you add truncated back propagation through time you will also get an additional argument with the hidden states of the previous step.

```
# Truncated back-propagation through time
def training_step(self, batch, batch_idx, hiddens):
    # hiddens are the hidden states from the previous truncated backprop step
    out, hiddens = self.lstm(data, hiddens)
    loss = ...
    return {"loss": loss, "hiddens": hiddens}
```

Note: The loss value shown in the progress bar is smoothed (averaged) over the last values, so it differs from the actual loss returned in train/validation step.

Note: When `accumulate_grad_batches` > 1, the loss returned here will be automatically normalized by `accumulate_grad_batches` internally.

---

validation_step(*batch*, *batch_idx*)

Operates on a single batch of data from the validation set. In this step you'd might generate examples or calculate anything of interest like accuracy.

```
# the pseudocode for these calls
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    val_outs.append(out)
validation_epoch_end(val_outs)
```

Parameters

- **batch** – The output of your `DataLoader`.

- **batch_idx** – The index of this batch.
- **dataloader_idx** – The index of the dataloader that produced this batch. (only if multiple val dataloaders used)

Returns

- Any object or value
- None - Validation will skip to the next batch

```
# pseudocode of order
val_outs = []
for val_batch in val_data:
    out = validation_step(val_batch)
    if defined("validation_step_end"):
        out = validation_step_end(out)
    val_outs.append(out)
val_outs = validation_epoch_end(val_outs)
```

```
# if you have one val dataloader:
def validation_step(self, batch, batch_idx):
    ...


# if you have multiple val dataloaders:
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    ...
```

Examples:

```
# CASE 1: A single validation dataset
def validation_step(self, batch, batch_idx):
    x, y = batch

    # implement your own
    out = self(x)
    loss = self.loss(out, y)

    # log 6 example images
    # or generated text... or whatever
    sample_imgs = x[:6]
    grid = torchvision.utils.make_grid(sample_imgs)
    self.logger.experiment.add_image('example_images', grid, 0)

    # calculate acc
    labels_hat = torch.argmax(out, dim=1)
    val_acc = torch.sum(y == labels_hat).item() / (len(y) * 1.0)

    # log the outputs!
    self.log_dict({'val_loss': loss, 'val_acc': val_acc})
```

If you pass in multiple val dataloaders, `validation_step()` will have an additional argument. We recommend setting the default value of 0 so that you can quickly switch between single and multiple dataloaders.

```
# CASE 2: multiple validation dataloaders
def validation_step(self, batch, batch_idx, dataloader_idx=0):
    # dataloader_idx tells you which dataset this is.
    ...
```

Note: If you don't need to validate you don't need to implement this method.

Note: When the `validation_step()` is called, the model has been put in eval mode and PyTorch gradients have been disabled. At the end of validation, the model goes back to training mode and gradients are enabled.

CHAPTER

TWENTYFOUR

SEMI-SUPERVISED LEARNING

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

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

24.1 Balanced classes

Example:

```
from pl_bolts.utils.semi_supervised import balance_classes
```

pl_bolts.utils.semi_supervised.**balance_classes**(*X*, *y*, *batch_size*)

Warning: The feature balance_classes is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Makes sure each batch has an equal amount of data from each class. Perfect balance.

Parameters

- **X** (Union[Tensor, ndarray]) – input features
- **y** (Union[Tensor, ndarray, Sequence[int]]) – mixed labels (ints)
- **batch_size** (int) – the ultimate batch size

Return type Tuple[ndarray, ndarray]

24.2 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*)

> **Warning:** The feature generate_half_labeled_batches is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Given a labeled dataset and an unlabeled dataset, this function generates a joint pair where half the batches are labeled and the other half is not.

Return type Tuple[ndarray, ndarray]

CHAPTER

TWENTYFIVE

SELF-SUPERVISED LEARNING

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

Note: We rely on the community to keep these updated and working. If something doesn't work, we'd really appreciate a contribution to fix!

25.1 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)
```

(continues on next page)

(continued from previous page)

```
# 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.modules.module.Module

> **Warning:** The feature FeatureMapContrastiveTask is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Performs an anchor, positive negative pair comparison for each each tuple of feature maps passed.

```
# extract feature maps
pos_0, pos_1, pos_2 = encoder(x_pos)
anc_0, anc_1, anc_2 = encoder(x_anchor)

# compare only the 0th feature maps
task = FeatureMapContrastiveTask('00')
loss, regularizer = task((pos_0), (anc_0))

# compare (pos_0 to anc_1) and (pos_0, anc_2)
task = FeatureMapContrastiveTask('01, 02')
losses, regularizer = task((pos_0, pos_1, pos_2), (anc_0, anc_1, anc_2))
loss = losses.sum()

# compare (pos_1 vs a anc_random)
task = FeatureMapContrastiveTask('0r')
loss, regularizer = task((pos_0, pos_1, pos_2), (anc_0, anc_1, anc_2))
```

```
# with bidirectional the comparisons are done both ways
task = FeatureMapContrastiveTask('01, 02')

# will compare the following:
# 01: (pos_0, anc_1), (anc_0, pos_1)
# 02: (pos_0, anc_2), (anc_0, pos_2)
```

Parameters

- **comparisons** (str) – groupings of feature map indices to compare (zero indexed, 'r' means random) ex: '00, 1r'
- **tclip** (float) – stability clipping value
- **bidirectional** (bool) – if true, does the comparison both ways

forward(*anchor_maps*, *positive_maps*)
Takes in a set of tuples, each tuple has two feature maps with all matching dimensions.

Example

```
>>> import torch
>>> from pytorch_lightning import seed_everything
>>> seed_everything(0)
0
>>> a1 = torch.rand(3, 5, 2, 2)
>>> a2 = torch.rand(3, 5, 2, 2)
>>> b1 = torch.rand(3, 5, 2, 2)
>>> b2 = torch.rand(3, 5, 2, 2)
...
>>> 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)]
```

25.2 Context prediction tasks

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

25.2.1 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.modules.module.Module

> **Warning:** The feature CPCTask is currently marked under review. The compatibility with other Lightning projects is not guaranteed and API may change at any time. The API and functionality may change without warning in future releases. More details: https://lightning-bolts.readthedocs.io/en/latest/stability.html

Loss used in CPC.

Initializes internal Module state, shared by both nn.Module and ScriptModule.

forward(*z*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

> **Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

CHAPTER

TWENTYSIX

CONTRIBUTING

Welcome to the PyTorch Lightning community! We're building the most advanced research platform on the planet to implement the latest, best practices that the amazing PyTorch team rolls out!

26.1 Bolts Design Principles

We encourage all sorts of contributions you're interested in adding! When coding for Bolts, please follow these principles.

26.1.1 Simple Internal Code

It's useful for users to look at the code and understand very quickly what's happening. Many users won't be engineers. Thus we need to value clear, simple code over condensed ninja moves. While that's super cool, this isn't the project for that :)

26.1.2 Force User Decisions To Best Practices

There are 1,000 ways to do something. However, eventually one popular solution becomes standard practice, and everyone follows. We try to find the best way to solve a particular problem, and then force our users to use it for readability and simplicity.

When something becomes a best practice, we add it to the framework. This is usually something like bits of code in utils or in the model file that everyone keeps adding over and over again across projects. When this happens, bring that code inside the trainer and add a flag for it.

26.1.3 Backward-compatible API

We all hate updating our deep learning packages because we don't want to refactor a bunch of stuff. In bolts, we make sure every change we make which could break an API is backward compatible with good deprecation warnings.

26.1.4 Gain User Trust

As a researcher, you can't have any part of your code going wrong. So, make thorough tests to ensure that every implementation of a new trick or subtle change is correct.

26.1.5 Interoperability

PyTorch Lightning Bolts is highly interoperable with PyTorch Lightning and PyTorch.

26.2 Contribution Types

We are always looking for help implementing new features or fixing bugs.

A lot of good work has already been done in project mechanics (requirements/base.txt, setup.py, pep8, badges, ci, etc…) so we're in a good state there thanks to all the early contributors (even pre-beta release)!

26.2.1 Bug Fixes:

1. If you find a bug please submit a GitHub issue.
    - Make sure the title explains the issue.
    - Describe your setup, what you are trying to do, expected vs. actual behaviour. Please add configs and code samples.
    - Add details on how to reproduce the issue - a minimal test case is always best, colab is also great. Note, that the sample code shall be minimal and if needed with publicly available data.
2. Try to fix it or recommend a solution. We highly recommend to use test-driven approach:
    - Convert your minimal code example to a unit/integration test with assert on expected results.
    - Start by debugging the issue… You can run just this particular test in your IDE and draft a fix.
    - Verify that your test case fails on the master branch and only passes with the fix applied.
3. Submit a PR!

***Note**, even if you do not find the solution, sending a PR with a test covering the issue is a valid contribution and we can help you or finish it with you :]*

26.2.2 New Features:

1. Submit a GitHub issue - describe what is the motivation of such feature (adding the use case or an example is helpful).
2. Let's discuss to determine the feature scope.
3. Submit a PR! We recommend test driven approach to adding new features as well:
    - Write a test for the functionality you want to add.
    - Write the functional code until the test passes.
4. Add/update the relevant tests!

- This PR is a good example for adding a new metric, and this one for a new logger.

26.2.3 New Models:

PyTorch Lightning Bolts makes several research models for ready usage. Following are general guidelines for adding new models.

1. Models which are standard baselines
2. Whose results are reproduced properly either by us or by authors.
3. Top models which are not SOTA but highly cited for production usage / for other uses. (E.g. Mobile BERT, MobileNets, FBNets).
4. Do not reinvent the wheel, natively support torchvision, torchtext, torchaudio models.
5. Use open source licensed models.

Please raise an issue before adding a new model. Please let us know why the particular model is important for bolts. There are tons of models that keep coming. It is very difficult to support every model.

26.2.4 Test cases:

Want to keep Lightning Bolts healthy? Love seeing those green tests? So do we! How to we keep it that way? We write tests! We value tests contribution even more than new features.

Tests are written using pytest. Tests in PyTorch Lightning bolts train model on a datamodule. Datamodule is lightning abstraction of representing dataloader and dataset. Model is checked by simply calling `.fit()` function over the datamodule.

Along with these we have tests for losses, callbacks and transforms as well.

Have a look at sample tests here.

After you have added the respective tests, you can run the tests locally with make script:

```
make test
```

Want to add a new test case and not sure how? Talk to us!

26.3 Note before submitting the PR, make sure you have run `pre-commit run`.

26.4 Guidelines

For this section, we refer to read the parent PL guidelines

Reminder

All added or edited code shall be the own original work of the particular contributor. If you use some third-party implementation, all such blocks/functions/modules shall be properly referred and if possible also agreed by code's author. For example - `This code is inspired from http://...`. In case you adding new dependencies, make sure that they are compatible with the actual PyTorch Lightning license (ie. dependencies should be *at least* as permissive as the PyTorch Lightning license).

26.4.1 Question & Answer

1. **How can I help/contribute?**

   All help is extremely welcome - reporting bugs, fixing documentation, adding test cases, solving issues and preparing bug fixes. To solve some issues you can start with label good first issue or chose something close to your domain with label help wanted. Before you start to implement anything check that the issue description that it is clear and self-assign the task to you (if it is not possible, just comment that you take it and we assign it to you...).

2. **Is there a recommendation for branch names?**

   We do not rely on the name convention so far you are working with your own fork. Anyway it would be nice to follow this convention `<type>/<issue-id>_<short-name>` where the types are: `bugfix`, `feature`, `docs`, `tests`, ...

3. **I have a model in other framework than PyTorch, how do I add it here?**

   Since PyTorch Lightning is written on top of PyTorch. We need models in PyTorch only. Also, we would need same or equivalent results with PyTorch Lightning after converting the models from other frameworks.

CHAPTER

TWENTYSEVEN

PL BOLTS GOVERNANCE | PERSONS OF INTEREST

27.1 Core Maintainers

- William Falcon (williamFalcon) (Lightning founder)
- Jirka Borovec (Borda)
- Akihiro Nitta (akihironitta)

27.2 Core Contributors

- Atharva Phatak (Atharva-Phatak)
- Shion Matsumoto (matsumotosan)
- JongMok Lee (lijm1358)

27.3 Alumni

- Ananya Harsh Jha (ananyahjha93)
- Annika Brundyn (annikabrundyn)
- Ota Jašek (otaj)
- Teddy Koker (teddykoker)

CHAPTER

TWENTYEIGHT

BOLTS STABILITY

Currently we are going through major revision of Bolts to ensure all of the code is stable and compatible with the rest of the Lightning ecosystem. For this reason, all of our features are either marked as stable or in need of review. Stable features are implicit, features to be reviewed are explicitly marked.

At the beginning of the aforementioned revision, **ALL** of the features currently in the project have been marked as to be reviewed and will undergo rigorous review and testing before they can be marked as stable. See this GitHub issue to check progress of the revision

This document is intended to help you know what to expect and to outline our commitment to stability.

28.1 Stable

For stable features, all of the following are true:

- the API isn't expected to change
- if anything does change, incorrect usage will give a deprecation warning for **one minor release** before the breaking change is made
- the API has been tested for compatibility with latest releases of PyTorch Lightning and Flash

28.2 Under Review

For features to be reviewed, any or all of the following may be true:

- the feature has unstable dependencies
- the API may change without notice in future versions
- the performance of the feature has not been verified
- the docs for this feature are under active development

Before a feature can be moved to Stable it needs to satisfy following conditions:

- Have appropriate tests, that will check not only correctness of the feature, but also compatibility with the current versions.
- Not have duplicate code accross Lightning ecosystem and more mature OSS projects.
- Pass a review process.

CHAPTER

TWENTYNINE

CHANGELOG

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

29.1 [unReleased] - 2022-MM-DD

29.1.1 [unReleased] - Added

-

29.1.2 [unReleased] - Changed

-

29.1.3 [unReleased] - Deprecated

-

29.1.4 [unReleased] - Removed

-

29.2 [0.7.0] - 2022-06-30

29.2.1 [0.7.0] - Added

- Improved YOLO model includes YOLOv4, YOLOv5, and YOLOX networks and training algorithms (#817)

29.2.2 [0.7.0] - Changed

- Move SSL transforms to pl_bolts/transforms (#905)
- Reviewed `models.detection.yolo` (#851)
- Reviewed `LogisticRegression` (#950)
- Bumped support of min python version to py3.8+ (#1021)
- Update `numpy` compatibility to <1.25.0 (#959)
- Update `torchmetrics` compatibility to <0.12.0 (#1016)
- Update `pytorch-lightning` compatibility to >1.7.0,<2.0.0 (#965, #973, #1006)

29.2.3 [0.7.0] - Fixed

- Dropped reference to `torch._six` (#993)

29.3 [0.6.0] - 2022-11-02

29.3.1 [0.6.0] - Added

- Updated SparseML callback for latest PyTorch Lightning (#822)
- Updated torch version to v1.10.X (#815)
- Dataset specific args method to CIFAR10, ImageNet, MNIST, and STL10 (#890)
- Migrate to use `lightning-utilities` (#907)
- Support PyTorch Lightning v1.8 (#910)
- Major revision of Bolts
  - `under_review` flag (#835, #837)
  - Reviewing GAN basics, `VisionDataModule`, `MNISTDataModule`, `CIFAR10DataModule` (#843)
  - Added tests, updated doc-strings for Dummy Datasets (#865)
  - Binary MNIST/EMNIST Datasets and Datamodules (#866)
  - FashionMNIST/EMNIST Datamodules (#871)
  - Revision `ArrayDataset` (#872)
  - BYOL weight update callback (#867)
  - Revision `models.vision.unet`, `models.vision.segmentation` (#880)
  - Revision of SimCLR transforms (#857)
  - Revision Metrics (#878, #887)
  - Revision of BYOL module and tests (#874)
  - Revision of MNIST module (#873)
  - Revision of dataset normalizations (#898)
  - Revision of SimSiam module and tests (#891)

  - Revision `datasets.kitti_dataset.KittiDataset` (#896)
  - SWAV improvements (#903)
  - minor dcgan-import fix (#921)

29.3.2 [0.6.0] - Fixed

- Removing extra flatten (#809)
- support number of channels!=3 in YOLOConfiguration (#806)
- CVE-2007-4559 Patch (#894)

29.4 [0.5.0] - 2021-12-20

29.4.1 [0.5.0] - Added

- Added YOLO model (#552)
- Added `SRGAN`, `SRImageLoggerCallback`, `TVTDataModule`, `SRCelebA`, `SRMNIST`, `SRSTL10` (#466)
- Added nn.Module support for FasterRCNN backbone (#661)
- Added `RetinaNet` with torchvision backbones (#529)
- Added Python 3.9 support (#786)

29.4.2 [0.5.0] - Changed

- VAE now uses deterministic KL divergence during training, previously estimated KL divergence by random sampling (#760)

29.4.3 [0.5.0] - Removed

- Removed PyTorch 1.6 support (#786)
- Removed Python 3.6 support (#785)

29.4.4 [0.5.0] - Fixed

- Fixed doctest fails with ImportError: cannot import name 'Env' from 'gym' (#751)
- Fixed MoCo v2 missing Cosine Annealing learning scheduler (#757)

29.5 [0.4.0] - 2021-09-09

29.5.1 [0.4.0] - Added

- Added Soft Actor Critic (SAC) Model (#627)
- Added `EMNISTDataModule`, `BinaryEMNISTDataModule`, and `BinaryEMNIST` dataset (#676)
- Added Advantage Actor-Critic (A2C) Model (#598)
- Added Torch ORT Callback (#720)
- Added SparseML Callback (#724)

29.5.2 [0.4.0] - Changed

- Changed the default values `pin_memory=False`, `shuffle=False` and `num_workers=16` to `pin_memory=True`, `shuffle=True` and `num_workers=0` of datamodules (#701)
- Supporting deprecated attribute usage (#699)

29.5.3 [0.4.0] - Fixed

- Fixed ImageNet val loader to use val transform instead of train transform (#713)
- Fixed the MNIST download giving HTTP 404 with `torchvision>=0.9.1` (#674)
- Removed momentum updating from val step and add separate val queue (#631)
- Fixed moving the queue to GPU when resuming checkpoint for SwAV model (#684)
- Fixed FP16 support with vision GPT model (#694)
- Removing bias from linear model regularisation (#669)
- Fixed CPC module issue (#680)

29.6 [0.3.4] - 2021-06-17

29.6.1 [0.3.4] - Changed

- Replaced `load_boston` with `load_diabetes` in the docs and tests (#629)
- Added base encoder and MLP dimension arguments to BYOL constructor (#637)

29.6.2 [0.3.4] - Fixed

- Fixed the MNIST download giving HTTP 503 (#633)
- Fixed type annotation of `ExperienceSource.__iter__` (#645)
- Fixed `pretrained_urls` on Windows (#652)
- Fixed logistic regression (#655, #664)
- Fixed double softmax in `SSLEvaluator` (#663)

29.7 [0.3.3] - 2021-04-17

29.7.1 [0.3.3] - Changed

- Suppressed missing package warnings, conditioned by `WARN_MISSING_PACKAGE="1"` (#617)
- Updated all scripts to LARS (#613)

29.7.2 [0.3.3] - Fixed

- Add missing `dataclass` requirements (#618)

29.8 [0.3.2] - 2021-03-20

29.8.1 [0.3.2] - Changed

- Renamed SSL modules: `CPCV2` >> `CPC_v2` and `MocoV2` >> `Moco_v2` (#585)
- Refactored *setup.py* to be typing friendly (#601)

29.9 [0.3.1] - 2021-03-09

29.9.1 [0.3.1] - Added

- Added Pix2Pix model (#533)

29.9.2 [0.3.1] - Changed

- Moved vision models (`GPT2`, `ImageGPT`, `SemSegment`, `UNet`) to `pl_bolts.models.vision` (#561)

29.9.3 [0.3.1] - Fixed

- Fixed BYOL moving average update (#574)
- Fixed custom gamma in rl (#550)
- Fixed PyTorch 1.8 compatibility issue (#580, #579)
- Fixed handling batchnorms in `BatchGradientVerification` (#569)
- Corrected `num_rows` calculation in `LatentDimInterpolator` callback (#573)

29.10 [0.3.0] - 2021-01-20

29.10.1 [0.3.0] - Added

- Added `input_channels` argument to UNet (#297)
- Added SwAV (#239, #348, #323)
- Added data monitor callbacks `ModuleDataMonitor` and `TrainingDataMonitor` (#285)
- Added DCGAN module (#403)
- Added `VisionDataModule` as parent class for `BinaryMNISTDataModule`, `CIFAR10DataModule`, `FashionMNISTDataModule`, and `MNISTDataModule` (#400)
- Added GIoU loss (#347)
- Added IoU loss (#469)
- Added semantic segmentation model `SemSegment` with `UNet` backend (#259)
- Added pption to normalize latent interpolation images (#438)
- Added flags to datamodules (#388)
- Added metric GIoU (#347)
- Added Intersection over Union Metric/Loss (#469)
- Added SimSiam model (#407)
- Added gradient verification callback (#465)
- Added Backbones to FRCNN (#475)

29.10.2 [0.3.0] - Changed

- Decoupled datamodules from models (#332, #270)
- Set PyTorch Lightning 1.0 as the minimum requirement (#274)
- Moved `pl_bolts.callbacks.self_supervised.BYOLMAWeightUpdate` to `pl_bolts.callbacks.byol_updates.BYOLMAWeightUpdate` (#288)
- Moved `pl_bolts.callbacks.self_supervised.SSLOnlineEvaluator` to `pl_bolts.callbacks.ssl_online.SSLOnlineEvaluator` (#288)
- Moved `pl_bolts.datamodules.*_dataset` to `pl_bolts.datasets.*_dataset` (#275)
- Ensured sync across val/test step when using DDP (#371)

- Refactored CLI arguments of models (#394)
- Upgraded DQN to use `.log` (#404)
- Decoupled DataModules from models - CPCV2 (#386)
- Refactored datamodules/datasets (#338)
- Refactored Vision DataModules (#400)
- Refactored `pl_bolts.callbacks` (#477)
- Refactored the rest of `pl_bolts.models.self_supervised` (#481, #479
- Update [`torchvision.utils.make_grid`(https://pytorch.org/docs/stable/torchvision/utils.html#torchvision.utils.make_grid)] kwargs to `TensorboardGenerativeModelImageSampler` (#494)

29.10.3 [0.3.0] - Fixed

- Fixed duplicate warnings when optional packages are unavailable (#341)
- Fixed `ModuleNotFoundError` when importing datamoules (#303)
- Fixed cyclic imports in `pl_bolts.utils.self_suprvised` (#350)
- Fixed VAE loss to use KL term of ELBO (#330)
- Fixed dataloders of `MNISTDataModule` to use `self.batch_size` (#331)
- Fixed missing `outputs` in SSL hooks for PyTorch Lightning 1.0 (#277)
- Fixed stl10 datamodule (#369)
- Fixes SimCLR transforms (#329)
- Fixed binary MNIST datamodule (#377)
- Fixed the end of batch size mismatch (#389)
- Fixed `batch_size` parameter for DataModules remaining (#344)
- Fixed CIFAR `num_samples` (#432)
- Fixed DQN `run_n_episodes` using the wrong environment variable (#525)

29.11 [0.2.5] - 2020-10-12

- Enabled PyTorch Lightning 1.0 compatibility

29.12 [0.2.4] - 2020-10-12

- Enabled manual returns (#267)

29.13 [0.2.3] - 2020-10-12

29.13.1 [0.2.3] - Added

- Enabled PyTorch Lightning 0.10 compatibility (#264)
- Added dummy datasets (#266)
- Added `KittiDataModule` (#248)
- Added `UNet` (#247)
- Added reinforcement learning models, losses and datamodules (#257)

29.14 [0.2.2] - 2020-09-14

- Fixed confused logit (#222)

29.15 [0.2.1] - 2020-09-13

29.15.1 [0.2.1] - Added

- Added pretrained VAE with resnet encoders and decoders
- Added pretrained AE with resnet encoders and decoders
- Added CPC pretrained on CIFAR10 and STL10
- Verified BYOL implementation

29.15.2 [0.2.1] - Changed

- Dropped all dependencies except PyTorch Lightning and PyTorch
- Decoupled datamodules from GAN (#206)
- Modularize AE & VAE (#196)

29.15.3 [0.2.1] - Fixed

- Fixed gym (#221)
- Fix L1/L2 regularization (#216)
- Fix max_depth recursion crash in AsynchronousLoader (#191)

29.16 [0.2.0] - 2020-09-10

29.16.1 [0.2.0] - Added

- Enabled Apache License, Version 2.0

29.16.2 [0.2.0] - Changed

- Moved unnecessary dependencies to `__main__` section in BYOL (#176)

29.16.3 [0.2.0] - Fixed

- Fixed CPC STL10 finetune (#173)

29.17 [0.1.1] - 2020-08-23

29.17.1 [0.1.1] - Added

- Added Faster RCNN + Pscal VOC DataModule (#157)
- Added a better lars scheduling `LARSWrapper` (#162)
- Added CPC finetuner (#158)
- Added `BinaryMNISTDataModule` (#153)
- Added learning rate scheduler to BYOL (#148)
- Added Cityscapes DataModule (#136)
- Added learning rate scheduler `LinearWarmupCosineAnnealingLR` (#138)
- Added BYOL (#144)
- Added `ConfusedLogitCallback` (#118)
- Added an asynchronous single GPU dataloader. (#1521)

29.17.2 [0.1.1] - Fixed

- Fixed simclr finetuner (#165)
- Fixed STL10 finetuner (#164)
- Fixed Image GPT (#108)
- Fixed unused MNIST transforms in tran/val/test (#109)

29.17.3 [0.1.1] - Changed

- Enhanced train batch function (#107)

29.18 [0.1.0] - 2020-07-02

29.18.1 [0.1.0] - Added

- Added setup and repo structure
- Added requirements
- Added docs
- Added Manifest
- Added coverage
- Added MNIST template
- Added VAE template
- Added GAN + AE + MNIST
- Added Linear Regression
- Added Moco2g
- Added simclr
- Added RL module
- Added Loggers
- Added Transforms
- Added Tiny Datasets
- Added regularization to linear + logistic models
- Added Linear and Logistic Regression tests
- Added Image GPT
- Added Recommenders module

29.18.2 [0.1.0] - Changed

- Device is no longer set in the DQN model init
- Moved RL loss function to the losses module
- Moved rl.common.experience to datamodules
- train_batch function to VPG model to generate batch of data at each step (POC)
- Experience source no longer gets initialized with a device, instead the device is passed at each step()
- Refactored ExperienceSource classes to be handle multiple environments.

29.18.3 [0.1.0] - Removed

- Removed N-Step DQN as the latest version of the DQN supports N-Step by setting the `n_step` arg to n
- Deprecated common.experience

29.18.4 [0.1.0] - Fixed

- Documentation
- Doct tests
- CI pipeline
- Imports and pkg
- CPC fixes

CHAPTER

THIRTY

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

allow_zero_length_dataloader_with_multiple_devices (*pl_bolts.datamodules.sklearn_datamodule.SklearnDataModule attribute*), 33
allow_zero_length_dataloader_with_multiple_devices (*pl_bolts.datamodules.ssl_imagenet_datamodule.SSLImagenetDataModule attribute*), 52

B

BinaryEMNISTDataModule (*class in pl_bolts.datamodules.binary_emnist_datamodule*), 37
BinaryMNISTDataModule (*class in pl_bolts.datamodules.binary_mnist_datamodule*), 38

C

CIFAR10DataModule (*class in pl_bolts.datamodules.cifar10_datamodule*), 42
CityscapesDataModule (*class in pl_bolts.datamodules.cityscapes_datamodule*), 40

D

dataset_cls (*pl_bolts.datamodules.binary_emnist_datamodule.BinaryEMNISTDataModule attribute*), 38
dataset_cls (*pl_bolts.datamodules.binary_mnist_datamodule.BinaryMNISTDataModule attribute*), 40
dataset_cls (*pl_bolts.datamodules.cifar10_datamodule.CIFAR10DataModule attribute*), 43
dataset_cls (*pl_bolts.datamodules.emnist_datamodule.EMNISTDataModule attribute*), 45
dataset_cls (*pl_bolts.datamodules.fashion_mnist_datamodule.FashionMNISTDataModule attribute*), 47
dataset_cls (*pl_bolts.datamodules.mnist_datamodule.MNISTDataModule attribute*), 51
default_transforms() (*pl_bolts.datamodules.binary_mnist_datamodule.BinaryMNISTDataModule method*), 40
default_transforms() (*pl_bolts.datamodules.cifar10_datamodule.CIFAR10DataModule method*), 43
default_transforms() (*pl_bolts.datamodules.emnist_datamodule.EMNISTDataModule method*), 45
default_transforms() (*pl_bolts.datamodules.fashion_mnist_datamodule.FashionMNISTDataModule method*), 47
default_transforms() (*pl_bolts.datamodules.mnist_datamodule.MNISTDataModule method*), 51

E

EMNISTDataModule (*class in pl_bolts.datamodules.emnist_datamodule*), 43

F

FashionMNISTDataModule (*class in pl_bolts.datamodules.fashion_mnist_datamodule*), 46

I

ImagenetDataModule (*class in pl_bolts.datamodules.imagenet_datamodule*), 48

M

MNISTDataModule (*class in pl_bolts.datamodules.mnist_datamodule*), 50

N

num_classes (*pl_bolts.datamodules.binary_mnist_datamodule.BinaryMNISTDataModule property*), 40
num_classes (*pl_bolts.datamodules.cifar10_datamodule.CIFAR10DataModule property*), 43
num_classes (*pl_bolts.datamodules.cityscapes_datamodule.CityscapesDataModule property*), 42
num_classes (*pl_bolts.datamodules.emnist_datamodule.EMNISTDataModule property*), 46
num_classes (*pl_bolts.datamodules.fashion_mnist_datamodule.FashionMNISTDataModule property*), 47

num_classes (*pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule property*), 50
num_classes (*pl_bolts.datamodules.mnist_datamodule.MNISTDataModule property*), 51

P

prepare_data() (*pl_bolts.datamodules.emnist_datamodule.EMNISTDataModule method*), 45
prepare_data() (*pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule method*), 49
prepare_data() (*pl_bolts.datamodules.ssl_imagenet_datamodule.SSLImagenetDataModule method*), 52
prepare_data() (*pl_bolts.datamodules.stl10_datamodule.STL10DataModule method*), 58
prepare_data_per_node (*pl_bolts.datamodules.sklearn_datamodule.SklearnDataModule attribute*), 33
prepare_data_per_node (*pl_bolts.datamodules.ssl_imagenet_datamodule.SSLImagenetDataModule attribute*), 52

S

setup() (*pl_bolts.datamodules.emnist_datamodule.EMNISTDataModule method*), 45
SklearnDataModule (*class in pl_bolts.datamodules.sklearn_datamodule*), 32
SSLImagenetDataModule (*class in pl_bolts.datamodules.ssl_imagenet_datamodule*), 52
STL10DataModule (*class in pl_bolts.datamodules.stl10_datamodule*), 56

T

test_dataloader() (*pl_bolts.datamodules.cityscapes_datamodule.CityscapesDataModule method*), 41
test_dataloader() (*pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule method*), 49
test_dataloader() (*pl_bolts.datamodules.sklearn_datamodule.SklearnDataModule method*), 33
test_dataloader() (*pl_bolts.datamodules.ssl_imagenet_datamodule.SSLImagenetDataModule method*), 53
test_dataloader() (*pl_bolts.datamodules.stl10_datamodule.STL10DataModule method*), 58
train_dataloader() (*pl_bolts.datamodules.cityscapes_datamodule.CityscapesDataModule method*), 41
train_dataloader() (*pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule method*), 49
train_dataloader() (*pl_bolts.datamodules.sklearn_datamodule.SklearnDataModule method*), 34
train_dataloader() (*pl_bolts.datamodules.ssl_imagenet_datamodule.SSLImagenetDataModule method*), 54
train_dataloader() (*pl_bolts.datamodules.stl10_datamodule.STL10DataModule method*), 58
train_dataloader_mixed() (*pl_bolts.datamodules.stl10_datamodule.STL10DataModule method*), 58
train_transform() (*pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule method*), 49

V

val_dataloader() (*pl_bolts.datamodules.cityscapes_datamodule.CityscapesDataModule method*), 42
val_dataloader() (*pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule method*), 49
val_dataloader() (*pl_bolts.datamodules.sklearn_datamodule.SklearnDataModule method*), 35
val_dataloader() (*pl_bolts.datamodules.ssl_imagenet_datamodule.SSLImagenetDataModule method*), 55
val_dataloader() (*pl_bolts.datamodules.stl10_datamodule.STL10DataModule method*), 58
val_dataloader_mixed() (*pl_bolts.datamodules.stl10_datamodule.STL10DataModule method*), 58
val_transform() (*pl_bolts.datamodules.imagenet_datamodule.ImagenetDataModule method*), 49