

An introduction to

Pinello Lab Journal Club



Lightning^{AI}



Michael Vinyard | September 14th, 2022



Today's Lightning tutorial

mvinyard / lightning-tutorial Public

Pin Unwatch 1 Fork 0


<> Code Issues Pull requests Actions Projects Wiki Security Insights Settings

main 1 branch 3 tags Go to file Add file <> Code

mvinyard Update README.md 97fbaad 7 minutes ago 43 commits

.github/workflows	add workflow	9 days ago
lightning_tutorial	compat	17 hours ago
notebooks	Update README.md	7 minutes ago
.gitignore	Initial commit	9 days ago
LICENSE	Initial commit	
README.md	Update README.md	
setup.py	update dep	

README.md

 **lightning-tutorial**

python 3.7 pypi package 0.0.0 code style black

Installation of the partner package

```
pip install lightning-tutorial
```

Table of contents

About

PyTorch-Lightning Tutorial

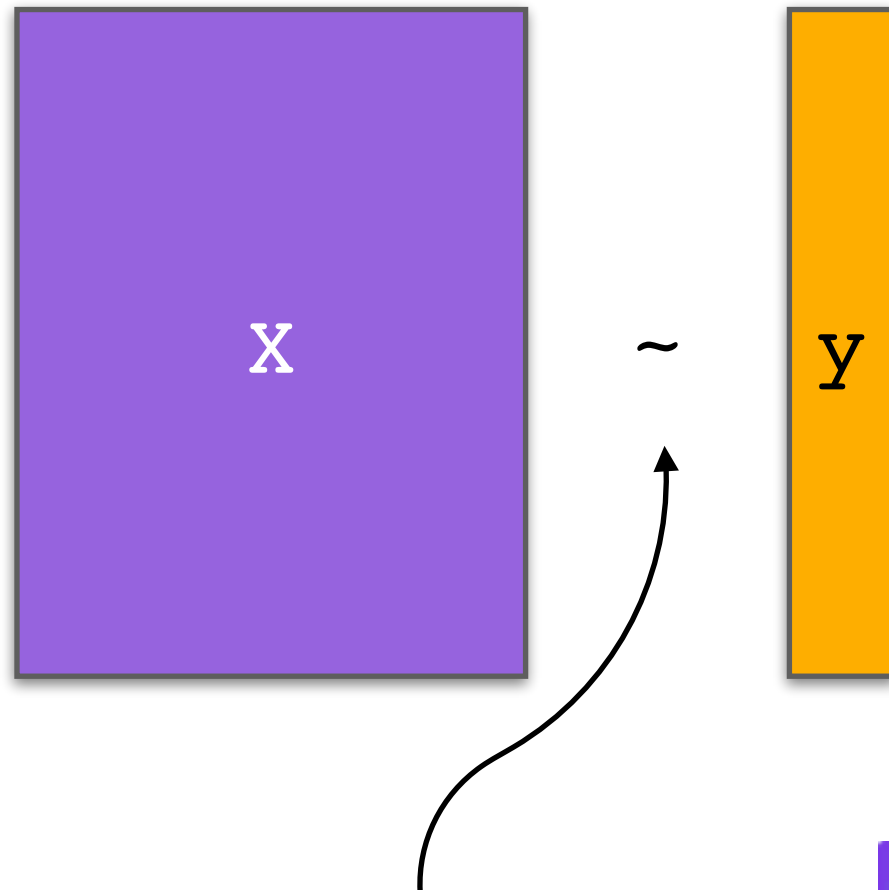
machine-learning tutorial deep-neural-networks ai deep-learning pytorch pytorch-lightning

Readme MIT license

Table of contents

- PyTorch Datasets and DataLoaders
 - Key module: `torch.utils.data.Dataset`
 - Key module: `torch.utils.data.DataLoader`
 - Other essential functions
- Single-cell data structures meet pytorch: `torch-adata`
- Lightning basics and the `LightningModule`
- `LightningDataModule`

Getting started



Some functional relationship
to be learned / represented
through a neural network

Loading and organizing data...

Defining a model...

Evaluating a model...

Iterating over various types of models...

Putting model into production and sharing...

You do the research.
Lightning will do everything else.



Source: [pytorch-lightning.readthedocs](https://pytorch-lightning.readthedocs.io)



Some PyTorch components to know

Tutorial

Previous situation

Before reading this article, your PyTorch script probably looked like this:

```
# Load entire dataset
X, y = torch.load('some_training_set_with_labels.pt')

# Train model
for epoch in range(max_epochs):
    for i in range(n_batches):
        # Local batches and labels
        local_X, local_y = X[i*n_batches:(i+1)*n_batches,], y[i*n_batches:(i+1)*n_batches,]

        # Your model
        [...]
```

or even this:

```
# Unoptimized generator
training_generator = SomeSingleCoreGenerator('some_training_set_with_labels.pt')

# Train model
for epoch in range(max_epochs):
    for local_X, local_y in training_generator:
        # Your model
        [...]
```

This article is about optimizing the entire data generation process, so that it does not become a bottleneck in the training procedure.

Source: <https://stanford.edu/~shervine/blog/pytorch-how-to-generate-data-parallel>



Some PyTorch components to know

`torch.utils.data.Dataset`

Dataset

- An overwrite-able python module
- Modify it at will!
- Must maintain the following **3 class methods**:

1. `__init__`
2. `__len__`
3. `__getitem__`

```
from torch.utils.data import Dataset
```

```
class TurtleData(Dataset):
```

```
    def __init__(self):
```

```
        """
        here we should pass requisite arguments
        that enable __len__() and __getitem__()
        """
```

```
    def __len__(self):
```

```
        """
        Returns the length/size/# of samples in the dataset.
        e.g., a 20,000 cell dataset would return `20_000`.
        """
        return # len
```

```
    def __getitem__(self, idx):
```

```
        """
        Subset and return a batch of the data.
```

```
        `idx` is the batch index (# of idx values = batch size).
        Maximum `idx` passed is <= `self.__len__()`
        """
```

```
        return # sampled data
```

These methods are named handles used under the hood by torch to get / pass data to models, etc.

Might seem constricting at first, but is actually quite liberating...



Some PyTorch components to know

`torch.utils.data.DataLoader`

DataLoader

- A “**base unit**” for data handling
- Similar to the usefulness of **AnnData**
- **Enables easy use of torch built-ins**

```
from torch.utils.data import DataLoader

dataset = TurtleData()
data_size = dataset.__len__()
print(data_size)
```

```
20_000
```

Other essential functions

```
from torch.utils.data import random_split


train_dataset, val_dataset = random_split(dataset, [18_000, 2_000])

# this can then be fed to a DataLoader, as above
train_loader = DataLoader(train_dataset)
val_loader = DataLoader(val_dataset)
```

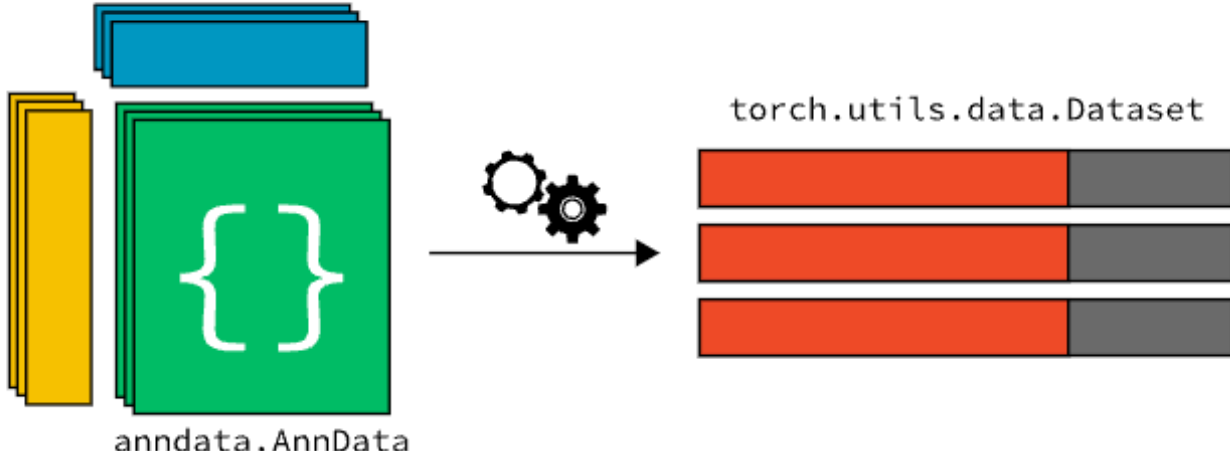


Some PyTorch components to know

☰ README.md



Create pytorch Datasets from `AnnData`



anndata.AnnData

torch.utils.data.Dataset

Example use of the base class

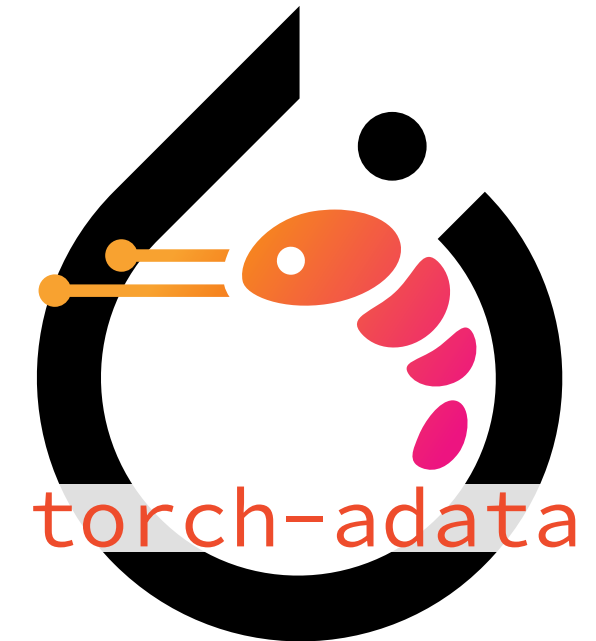
The base class, `AnnDataset` is a subclass of the widely-used `torch.utils.data.Dataset`.

```
import anndata as a
import torch_adata

adata = a.read_h5ad("/path/to/data.h5ad")
dataset = torch_adata.AnnDataset(adata)
```

Returns sampled data `X_batch` as a `torch.Tensor`.

```
# create a dummy index
idx = np.random.choice(range(dataset.__len__()), 5)
X_batch = dataset.__getitem__(idx)
```



PyTorch Lightning value offer

```
def training_step(self, batch):  
    x, y = batch  
    z = self.encoder(x)  
    x_hat = self.decoder(z)  
    mse = F.mse_loss(x_hat, x)  
    reg = self.discriminator(x_hat)  
    loss = mse + reg  
    return loss
```

Full flexibility

Try any ideas using raw PyTorch without the boilerplate.

```
if gpu:  
x = x.cuda(0)  
z = encoder(x)  
x_hat = decoder(z)  
.backward()
```






Reproducible + Readable

Decoupled research and engineering code enable reproducibility and better readability.

```
Trainer(  
    devices=32,  
    accelerator='gpu|tpu|hpu'  
)
```

Simple multi-GPU training

Use multiple GPUs/TPUs/HPUs etc... without code changes.

- ✓  PL.pytorch-lightning (GPUs)
- ✓  pre-commit.ci - pr — Succes
- ✓  PL.pytorch-lightning (HPUs)
- ✓  Milestone Check — Great, the
- ✓  PL.pytorch-lightning (IPUs)

Built-in testing

We've done all the testing so you don't have to.



Lightning enables code organization

Lightning enables you to organize your code
into **3 general buckets**



Data

Modularity

Model

What you want to
spend time on!

Callbacks

Visualization, etc., ...

Reproducible and readable

+ Automatic compatibility with various devices
(GPUs, Apple Silicon, TPUs, HPUs, etc...)



Lightning enables code organization

The screenshot displays a GitHub repository for `mvinyard / sc-neural-diffeqs`. The repository is private and has 0 watches, 0 forks, and 0 stars. The main branch is selected, showing 3 branches and 0 tags. The file browser overlay is open, showing the following structure:

- `mvinyard` update setup.py
- `.github/workflows` workflow for
- `analysis` add paper a
- `docs/images` updated log
- `scdiffeq.egg-info` Merge bran
- `scdiffeq` update Bas
- `.gitignore` adding gitig
- `LICENSE` Update LIC
- `README.md` Update REA
- `setup.py` Update setu

The overlay also shows the following files and folders:

- `_base_ancilliary` adding fate bias loss
- `_lightning_callbacks` temporary (messy) solutions to fix #12
- `_BaseModel.py` update BaseModel

The repository's README.md is visible at the bottom, featuring the `scdiffeq` logo and a badge indicating the package is available on PyPI (0.0.42) and uses Black for code style.



LightningDataModule

pytorch_lightning

```
from pytorch_lightning import LightningModule

class YourSOTAModel(LightningModule):
    def __init__(self,
                  net,
                  optimizer_kwargs={"lr":1e-3},
                  scheduler_kwargs={},
                  ):
        super().__init__()

        self.net = net
        self.optimizer_kwargs = optimizer_kwargs
        self.scheduler_kwargs = scheduler_kwargs

    def forward(self, batch):

        x, y = batch

        y_hat = self.net(x)
        loss = LossFunc(y_hat, y)

        return y_hat, loss

    def training_step(self, batch, batch_idx):

        y_hat, loss = self.forward(batch)

        return loss.sum()

    def validation_step(self, batch, batch_idx):

        y_hat, loss = self.forward(batch)

        return loss.sum()

    def test_step(self, batch, batch_idx):

        y_hat, loss = self.forward(batch)

        return loss.sum()

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters())
        scheduler = torch.optim.lr_scheduler.StepLR(optimizer(), **self._scheduler_kwargs)

        return [optimizer, ...], [scheduler, ...]
```

```
def validation_step(self, batch, batch_idx):

    y_hat, loss = self.forward(batch)

    return loss.sum()

def test_step(self, batch, batch_idx):

    y_hat, loss = self.forward(batch)

    return loss.sum()
```

```
def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.parameters(), **self._optim_kwargs)
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer(), **self._scheduler_kwargs)

    return [optimizer, ...], [scheduler, ...]
```

LightningDataModules

Why do I need a DataModule?

In normal PyTorch code, the data cleaning/preparation is usually scattered across many files. This makes sharing and reusing the exact splits and transforms across projects impossible.

Datamodules are for you if you ever asked the questions:

- what splits did you use?
- what transforms did you use?
- what normalization did you use?
- how did you prepare/tokenize the data?

`pytorch_lightning.LightningDataModule`



LightningDataModules

PyTorch

```
class MNISTClassifier(nn.Module):

    def __init__(self):
        self.layer_1 = torch.nn.Linear(28 * 28, 128)
        self.layer_2 = torch.nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.layer_1(x)
        x = F.relu(x)
        x = self.layer_2(x)
        return x

# download data
if global_rank == 0:
    mnist_train = MNIST(os.getcwd(), train=True, download=True)
    mnist_test = MNIST(os.getcwd(), train=False, download=True)

dist.barrier()

# transforms
transform=transforms.Compose([transforms.ToTensor(),
                              transforms.Normalize((0.1307,), (0.3081,))])
mnist_train = MNIST(os.getcwd(), train=True, transform=transform)
mnist_test = MNIST(os.getcwd(), train=False, transform=transform)

# split dataset
mnist_train, mnist_val = random_split(mnist_train, [55000, 5000])
mnist_test = MNIST(os.getcwd(), train=False, download=True)

# build dataloaders
mnist_train = DataLoader(mnist_train, batch_size=64)
mnist_val = DataLoader(mnist_val, batch_size=64)
mnist_test = DataLoader(mnist_test, batch_size=64)
pytorch_model = MNISTClassifier()
optimizer = torch.optim.Adam(pytorch_model.parameters(), lr=1e-3)

def cross_entropy_loss(logits, labels):
    return F.nll_loss(logits, labels)

num_epochs = 1
for epoch in range(num_epochs):
    for train_batch in mnist_train:
        x, y = train_batch
        logits = pytorch_model(x)
        loss = cross_entropy_loss(logits, y)
        print('train loss: ', loss.item())
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    with torch.no_grad():
        val_loss = []
        for val_batch in mnist_val:
            x, y = val_batch
            logits = pytorch_model(x)
            val_loss.append(cross_entropy_loss(logits, y).item())
```

LightningDataModules: LARRY

README.md

LARRY dataset

python 3.7 pypi package 0.0.0 code style black

Installation

pip distribution

```
pip install larry-dataset
```

Development version

```
git clone https://github.com/mvinyard/LARRY-dataset.git; cd LARRY-dataset
pip install -e .
```

Quickstart

Downloads pre-processed data from [AllonKleinLab/paper-data](#) to `./KleinLabData` (by default). The data is formatted into `AnnData` and returned to the user. A `.h5ad` file is also saved, locally. The data downloading and conversion step take several minutes due to the large expression `normed_counts` matrices though this only happens once.

```
import larry

dataset = "in_vitro" # can also choose from: "in_vivo" or "cytokine_perturbation"
adata = larry.fetch(dataset)
```

```
AnnData object with n_obs × n_vars = 130887 × 25289
  obs: 'Library', 'Cell barcode', 'Time point', 'Starting population', 'Cell type annotation', 'W
  var: 'gene_name'
  obsm: 'X_clone'
```

