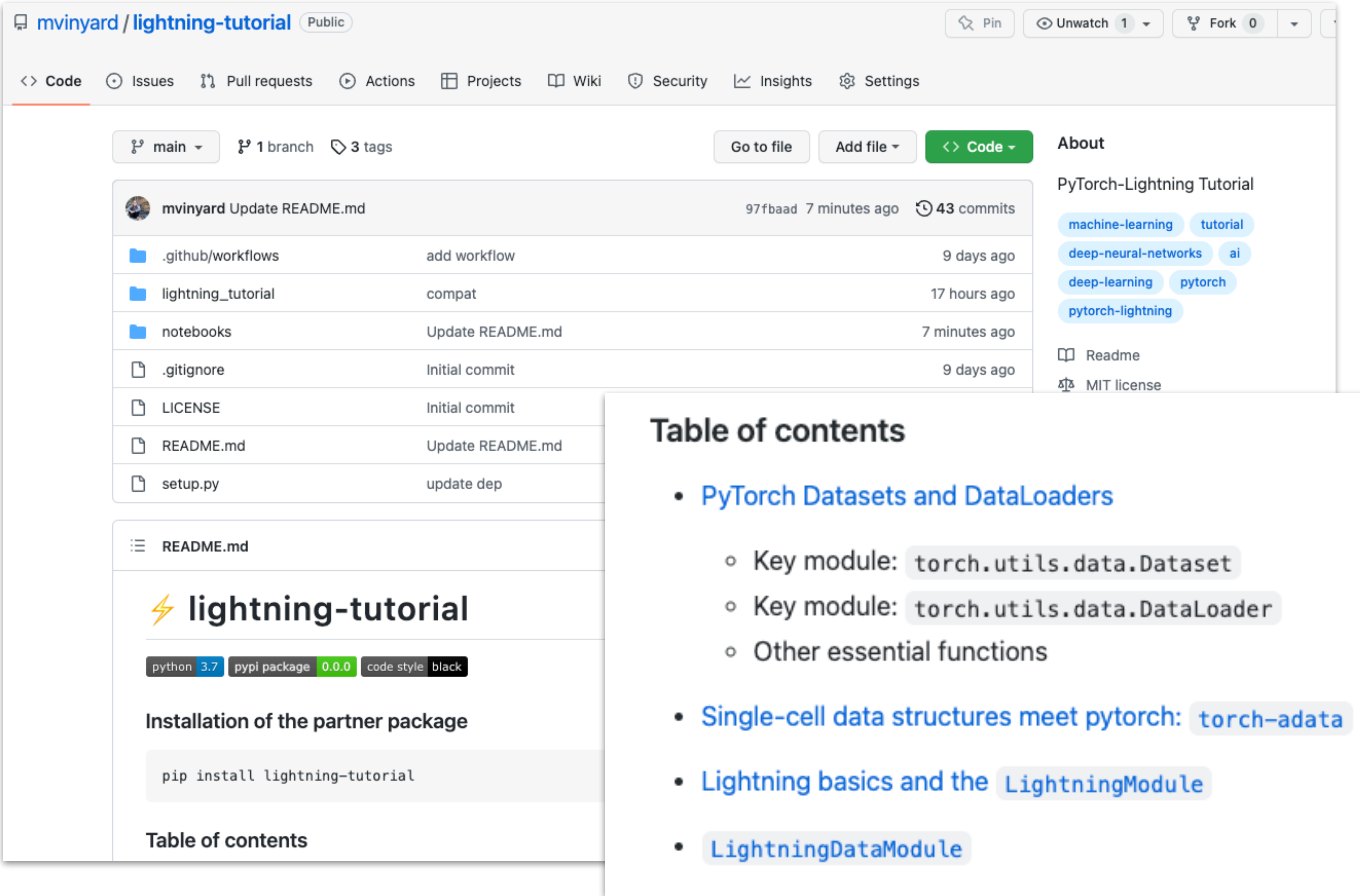
An introduction to Lightning AI

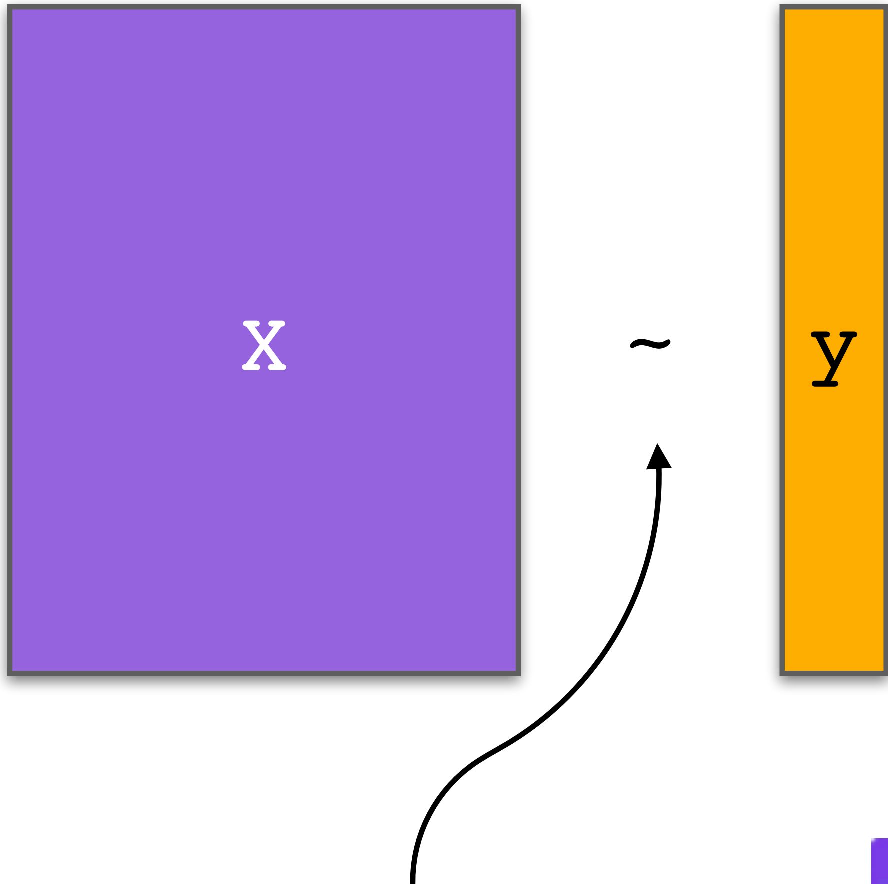
Pinello Lab Journal Club

Michael Vinyard | September 14th, 2022

Today's Lightning tutorial



Getting started



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

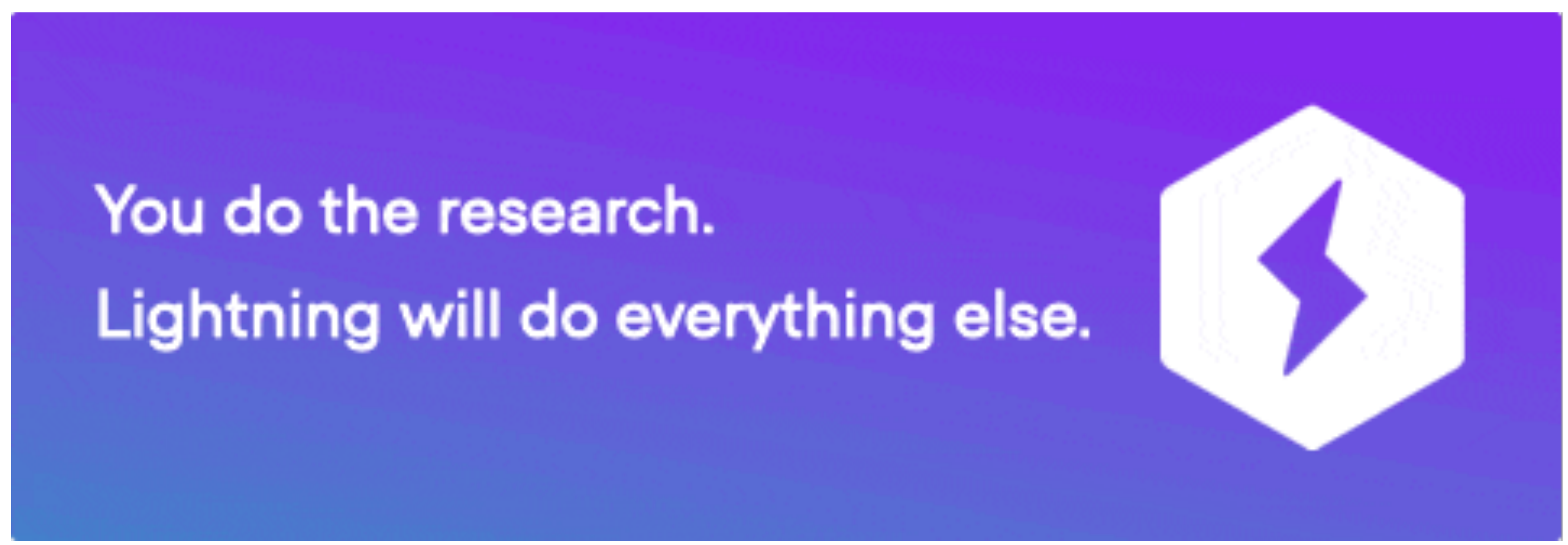
Loading and organizing data…

Defining a model…

Evaluating a model…

Iterating over various types of models…

Putting model into production and sharing…



Source: `pytorch-lightning.readthedocs`

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

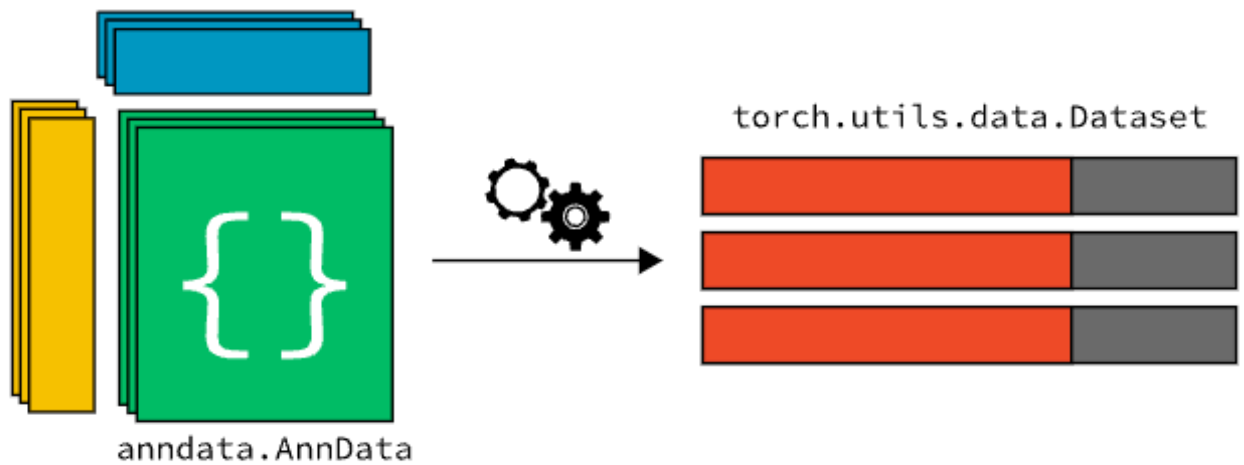
```
from torch.utils.data import random_split

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

torch-adata

Create pytorch Datasets from `AnnData`



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

```python
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.

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

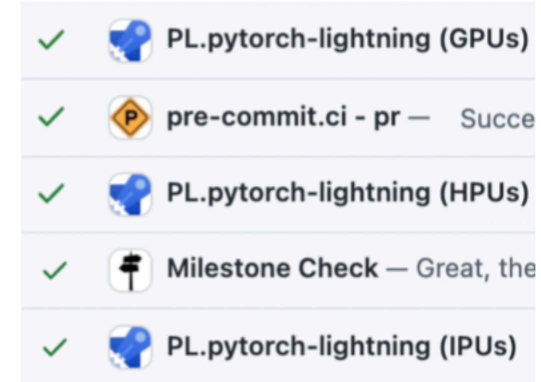
Reproducible + Readable

Decoupled research and engineering code enable reproducibility and better readability.

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

Simple multi-GPU training

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



Built-in testing

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

Images from `pytorch-lightning.readthedocs`

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!

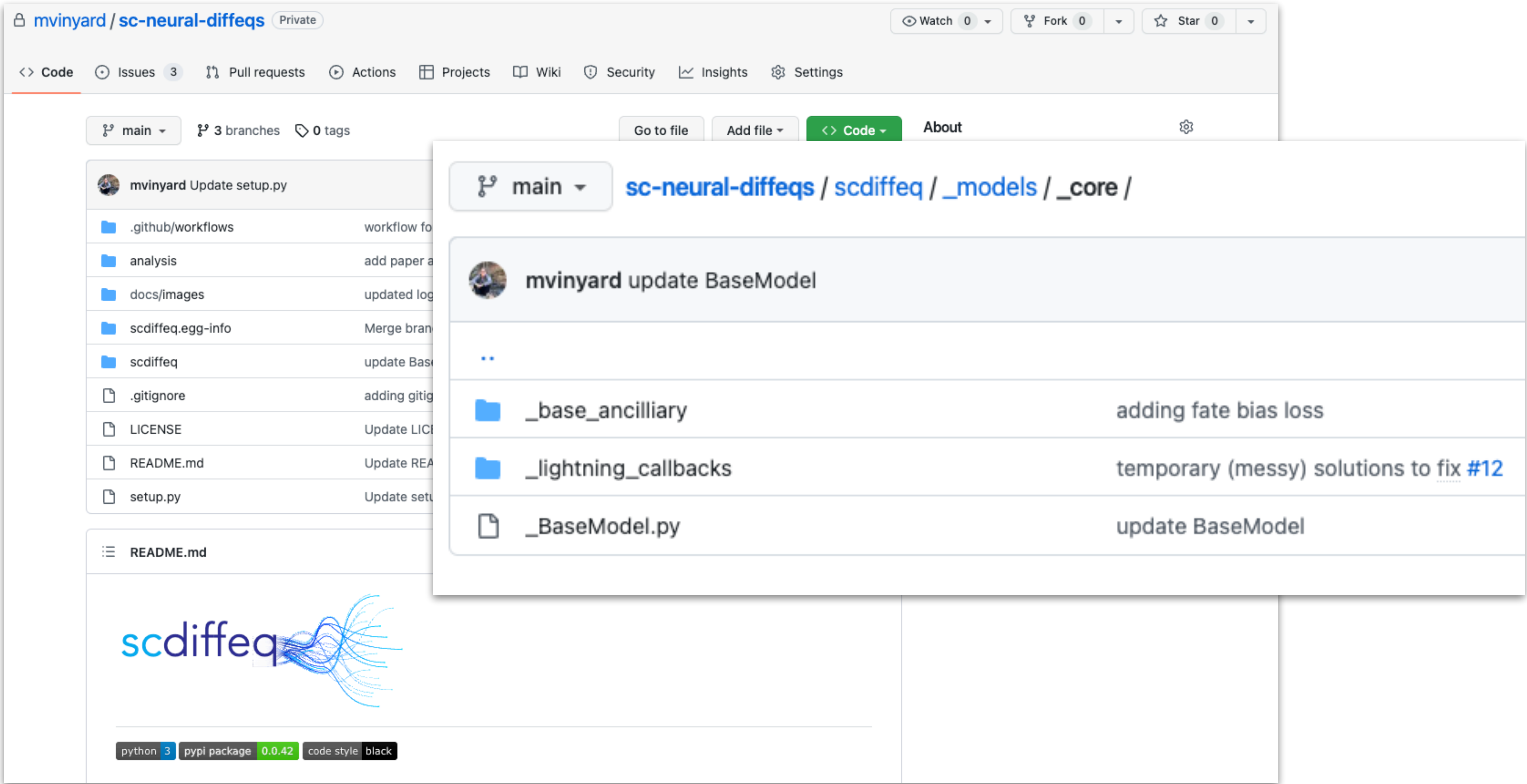
Callbacks

Visualization, etc., …

Reproducible and readable

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

Lightning enables code organization



LightningDataModules

pytorch_li

```python
from pytorch_lightning imoport 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.param
        scheduler = torch.optim.lr_scheduler.St

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

```python
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()
```

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'
```