Neural Latents Benchmark with PyTorch Lightning

Toolbox Creator: Andrew Sedler

Presenter: Chris Versteeg

NLB 2022

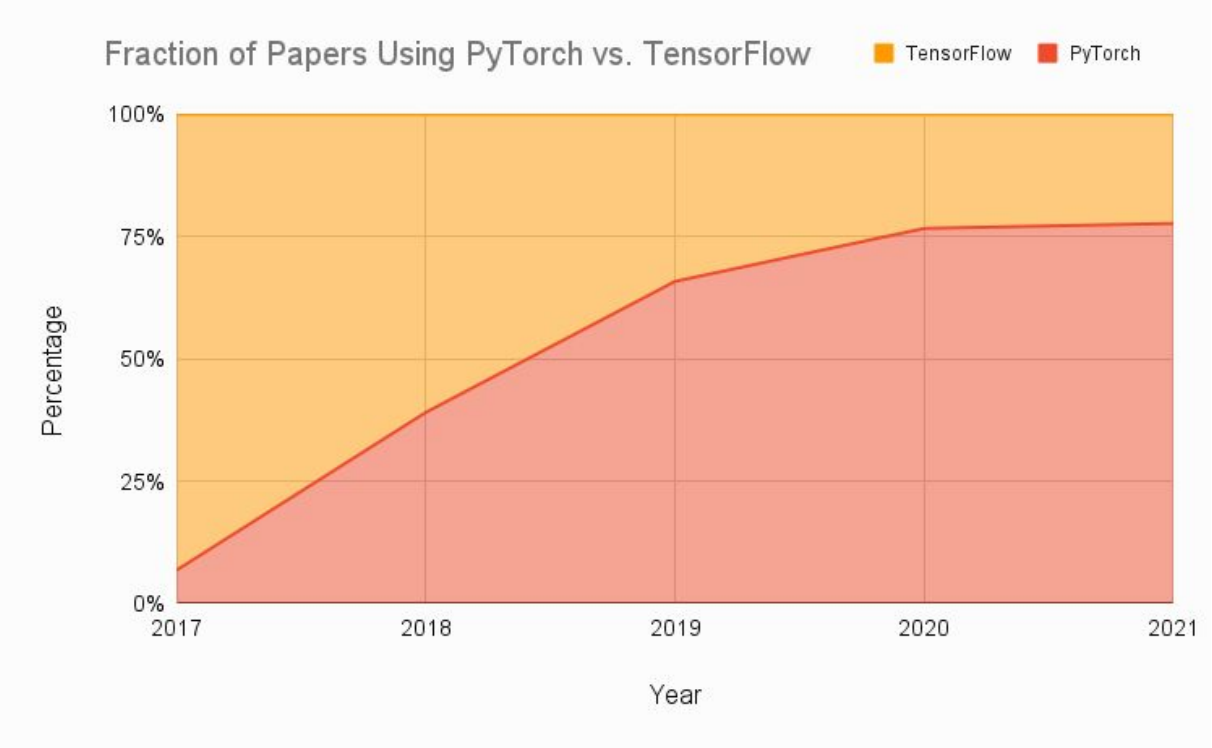
02/27/2022

Goal: Code a submission to NLB using PyTorch Lightning

- What is PyTorch?
- Why PyTorch Lightning?
- Overview of PyTorch Lightning
- Submitting your model to NLB competition using nlb-lightning

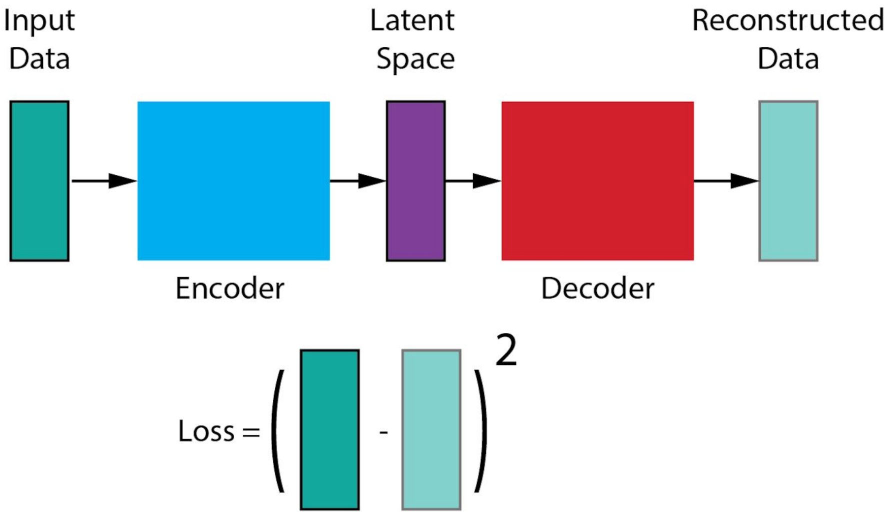
PyTorch: Modern deep learning framework for python

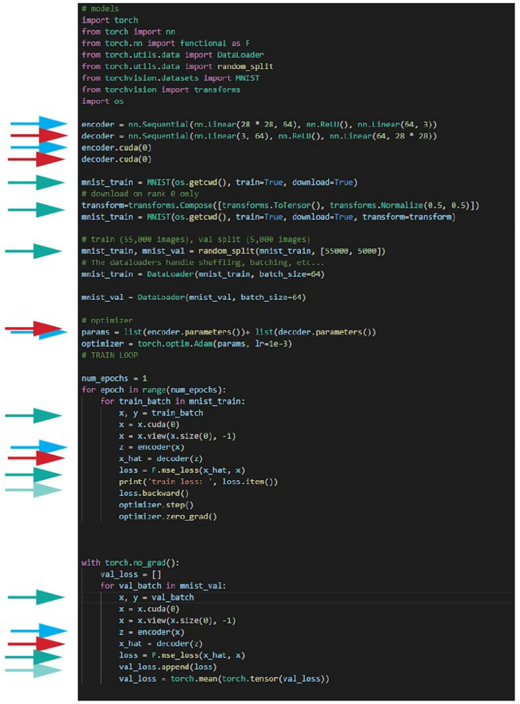
- The goal of deep learning research is to answer a question.
  - But you need to be able to build models!
- Most researcher don't need to concern themselves with the complex implementation details
- PyTorch abstracts away low-level engineering so you can focus on your research question.
- Alternatives: TensorFlow, etc.



https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2022/

Example: Building an autoencoder with Vanilla PyTorch



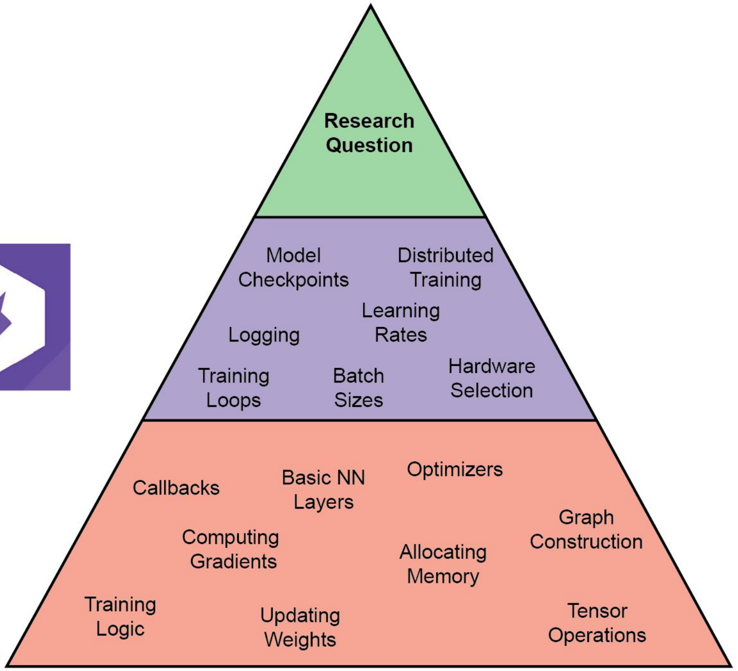


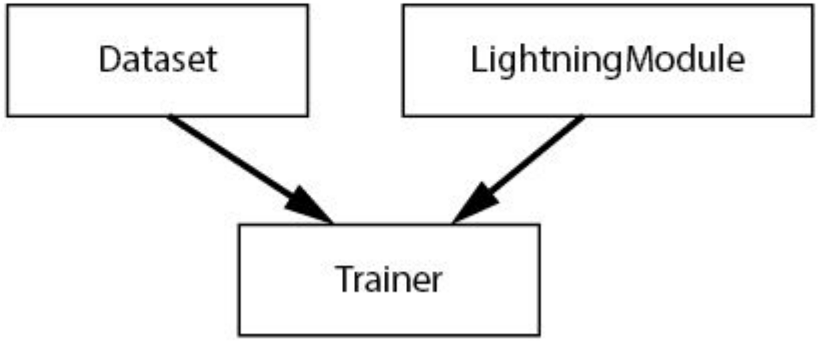
```python
# models
import torch
from torch import nn
from torch.nn import functional as F
from torch.utils.data import DataLoader
from torch.utils.data import random_split
from torchvision.datasets import MNIST
from torchvision import transforms
import os

encoder = nn.Sequential(nn.Linear(28 * 28, 64), nn.ReLU(), nn.Linear(64, 3))
decoder = nn.Sequential(nn.Linear(3, 64), nn.ReLU(), nn.Linear(64, 28 * 28))
encoder.cuda(0)
decoder.cuda(0)

mnist_train = MNIST(os.getcwd(), train=True, download=True)
# download on rank 0 only
transform=transforms.Compose([transforms.ToTensor(), transforms.Normalize(0.5, 0.5)])
mnist_train = MNIST(os.getcwd(), train=True, download=True, transform=transform)

# train (55,000 images), val split (5,000 images)
mnist_train, mnist_val = random_split(mnist_train, [55000, 5000])
# The dataloaders handle shuffling, batching, etc...
mnist_train = DataLoader(mnist_train, batch_size=64)

mnist_val = DataLoader(mnist_val, batch_size=64)

# optimizer
params = list(encoder.parameters())+ list(decoder.parameters())
optimizer = torch.optim.Adam(params, lr=1e-3)
# TRAIN LOOP

num_epochs = 1
for epoch in range(num_epochs):
    for train_batch in mnist_train:
        x, y = train_batch
        x = x.cuda(0)
        x = x.view(x.size(0), -1)
        z = encoder(x)
        x_hat = decoder(z)
        loss = F.mse_loss(x_hat, x)
        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
        x = x.cuda(0)
        x = x.view(x.size(0), -1)
        z = encoder(x)
        x_hat = decoder(z)
        loss = F.mse_loss(x_hat, x)
        val_loss.append(loss)
        val_loss = torch.mean(torch.tensor(val_loss))
```

Problem: Code is spread across the main function, so it isn't readily modular, reproducible or shareable.

PyTorch Lightning extends PyTorch

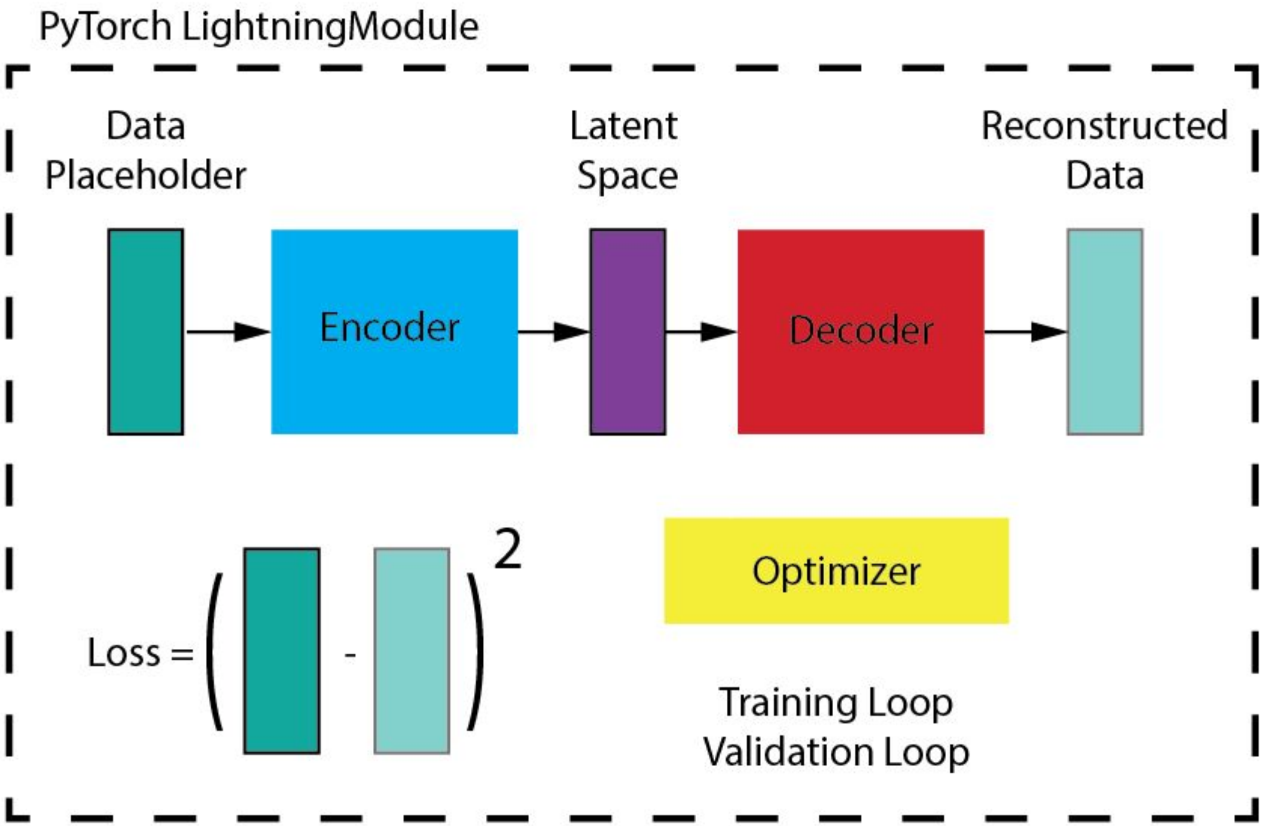
- Additional abstraction
  - Training/ Validation Loops
  - Switching to/from GPUs
  - Setting training flags
- Allows you to focus on your research question, not the engineering!
- Object-oriented development framework
  - All methods needed to train and test a model combined into a single object
- Three components
  - LightningModule
  - Dataset
  - Trainer

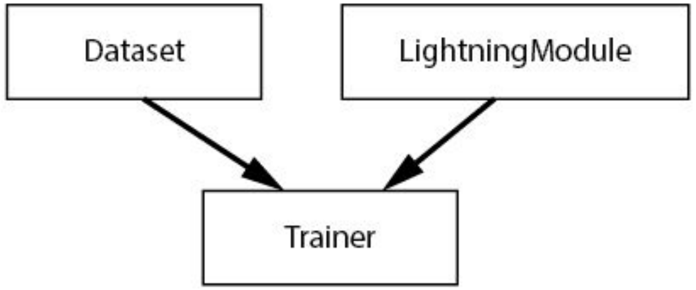




LightningModule

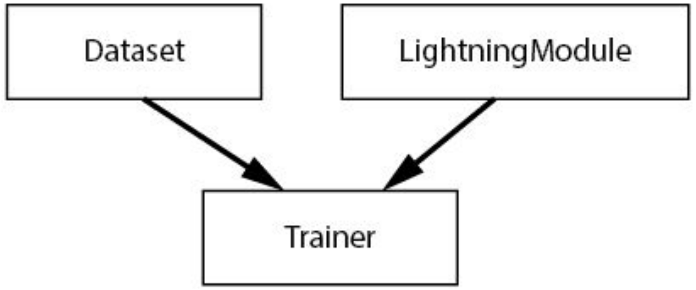
- Inherits from nn.Module
- Inside:
  - Model architecture
  - Loss function
  - Optimizer
  - Training/ Validation loops
- To train, simply pass LightningModule and Dataset into Trainer!

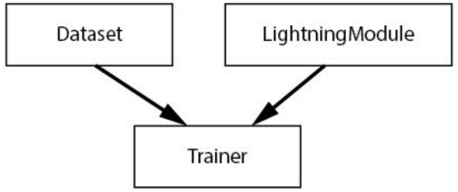




Dataset

- Potential forms:
  - PyTorch DataLoader
  - LightningDataModule
- LightningDataModule
  - Preprocesses data
  - Splits, transforms, augmentation
- Modularity and reproducibility!

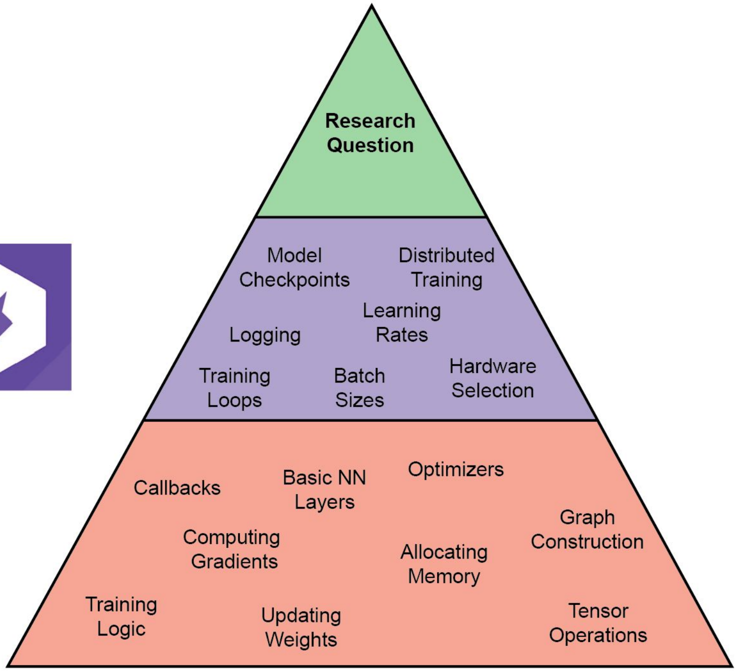


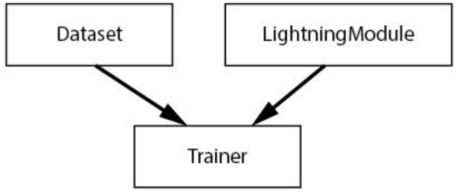


Trainer

- Object in PTL module
- Takes in training parameters
  - GPU/CPU settings
  - Numerical precision
  - Gradient accumulation/clipping
  - Automated batching
  - Loggers/ Callbacks
  - More!
- Abstracts:
  - Gradient handling
  - Running training, test, val
  - Callbacks
  - Device handling

```
# training
trainer = pl.Trainer(gpus=4, num_nodes=8, precision=16, limit_train_batches=0.5)
trainer.fit(model, train_loader, val_loader)
```

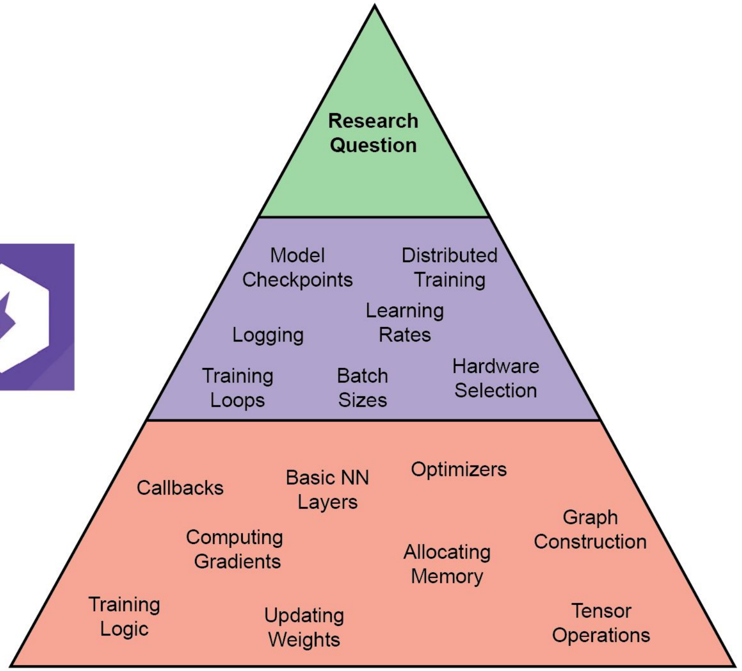




Trainer

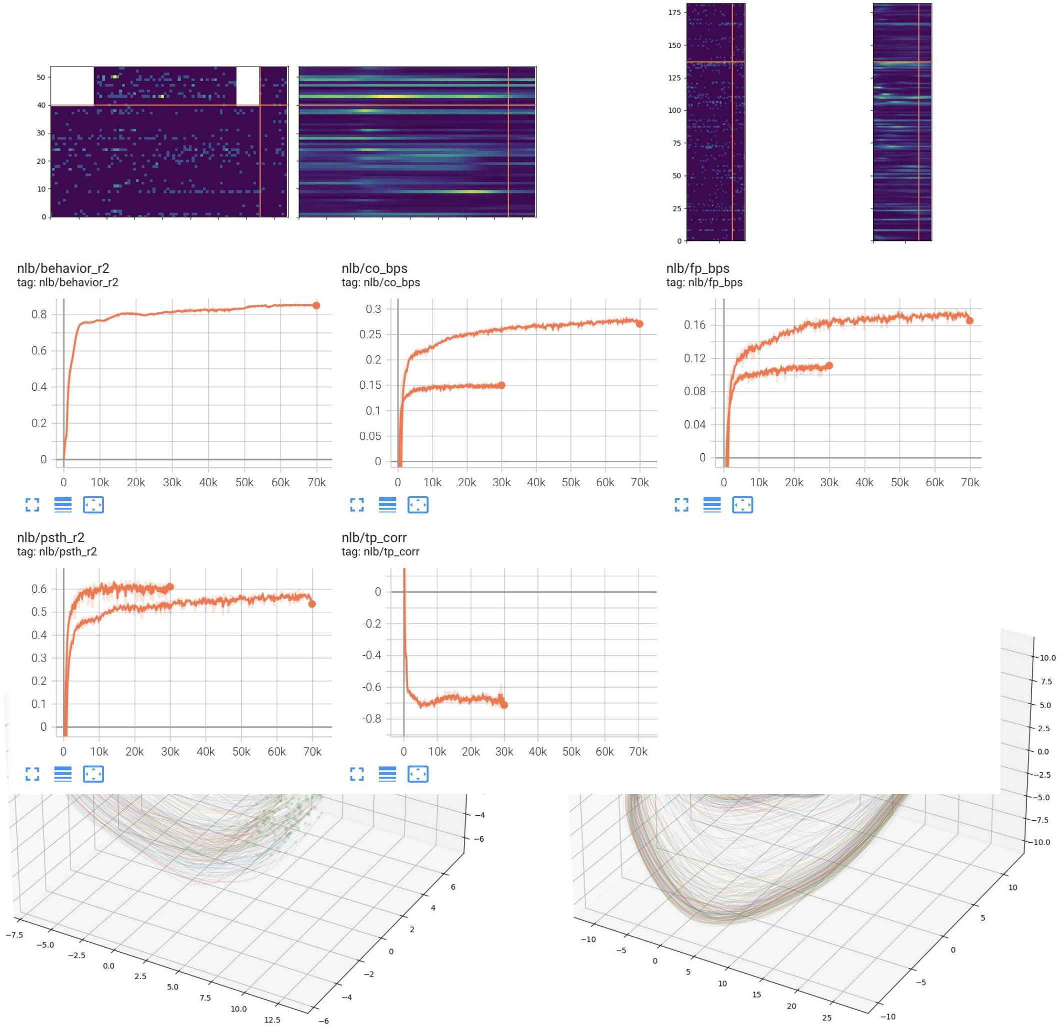
- Object in PTL module
- Takes in training parameters
  - GPU/CPU settings
  - Numerical precision
  - Gradient accumulation/clipping
  - Automated batching
  - Loggers/ **Callbacks**
  - More!
- Abstracts:
  - Gradient handling
  - Running training, test, val
  - Callbacks
  - Device handling

```
# training
trainer = pl.Trainer(gpus=4, num_nodes=8, precision=16, limit_train_batches=0.5)
trainer.fit(model, train_loader, val_loader)
```



Callbacks simplify model tuning and visualization

- Benefits of callbacks
  - Run at specific location
    - on_epoch_end
    - on_init_start
  - Arbitrary modification of the training/testing procedure
    - Early stopping
    - Learning rate modification
  - Visualize model performance
    - Rasters
    - Metrics
- Useful callbacks implemented in NLB Lightning repo:



Modular, reusable way to modify your models

Building your own NLB submission using nlb-lightning

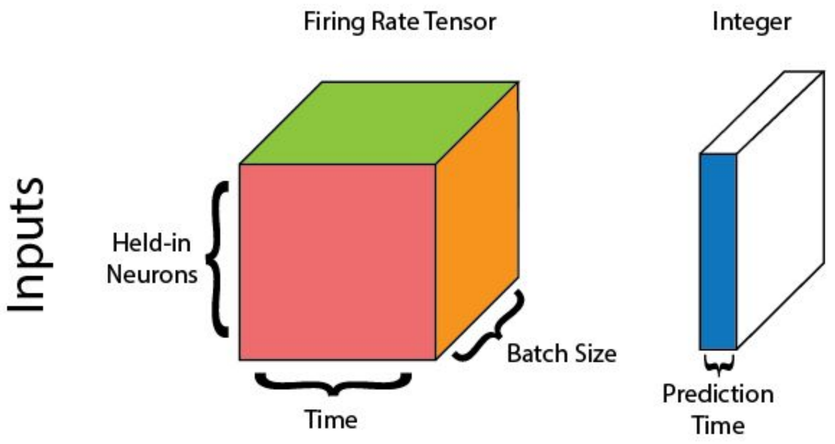
Example submission to NLB with PyTorch Lightning

- https://github.com/arsedler9/nlb-lightning
- Contents:
  - DataModule for NLB dataset
  - Example LightningModule
  - Example training script
  - Function to generate submission
  - Callbacks
- Steps:
  - Download data
  - Set up environment
  - Run “preprocess.py” to generate dataset
  - Run “train_sae.py” to train model and generate submission file
- Upload submission-{phase}.h5 to EvalAI!

Rank	Participant team		co-bps (↑)	vel R2 (↑)	psth R2 (↑)	fp-bps (↑)
1	AE Studio. (AESMTE3 [Ensemble])		0.3676	0.9114	0.6683	0.2589
2	AE Studio (AESMTE1)		0.3599	0.9105	0.6641	0.2470
3	Hennequin Lab (iLQR-VAE)		0.3559	0.8840	0.6062	0.1480
4	Neural Latents (AutoLFADS)	B	0.3364	0.9097	0.6360	0.2349
5	Churchland Lab (MINT)	B	0.3304	0.9121	0.7496	0.2076
6	Neural Latents (NDT)	B	0.3229	0.8862	0.5308	0.2206
7	NCLab		0.3039	0.7581	-1.0294	-0.0061
8	Neural Latents (SLDS)	B	0.2249	0.7947	0.5330	-1.1579

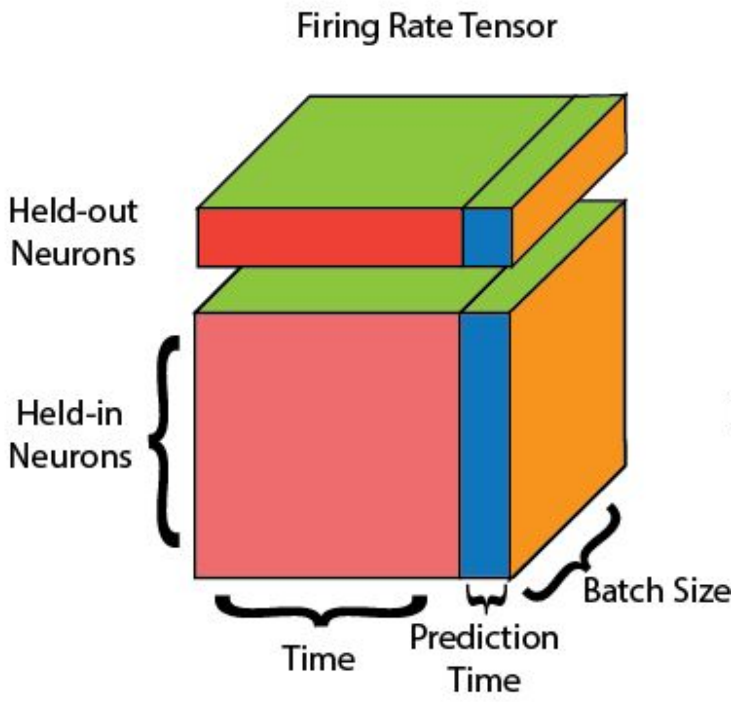
Requirements to train your own model (pt 1)

- To use nlb-lightning your model must inherit from LightningModule
- Forward() must have
  - Two inputs
  - Two outputs



Requirements to train your own model (pt 2)

- Validation step should expect training tensors in order
  - heldin
  - heldin_forward
  - heldout
  - heldout_forward
  - behavior
- Test step should only expect heldin
- See train_sae.py for a useful template
- train_sae_all.py supports training models in parallel



Questions?

See us at the poster session!