PPGEEC2318

Machine Learning

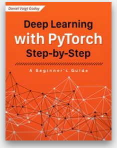
Rethinking the training loop: a simple classification problem

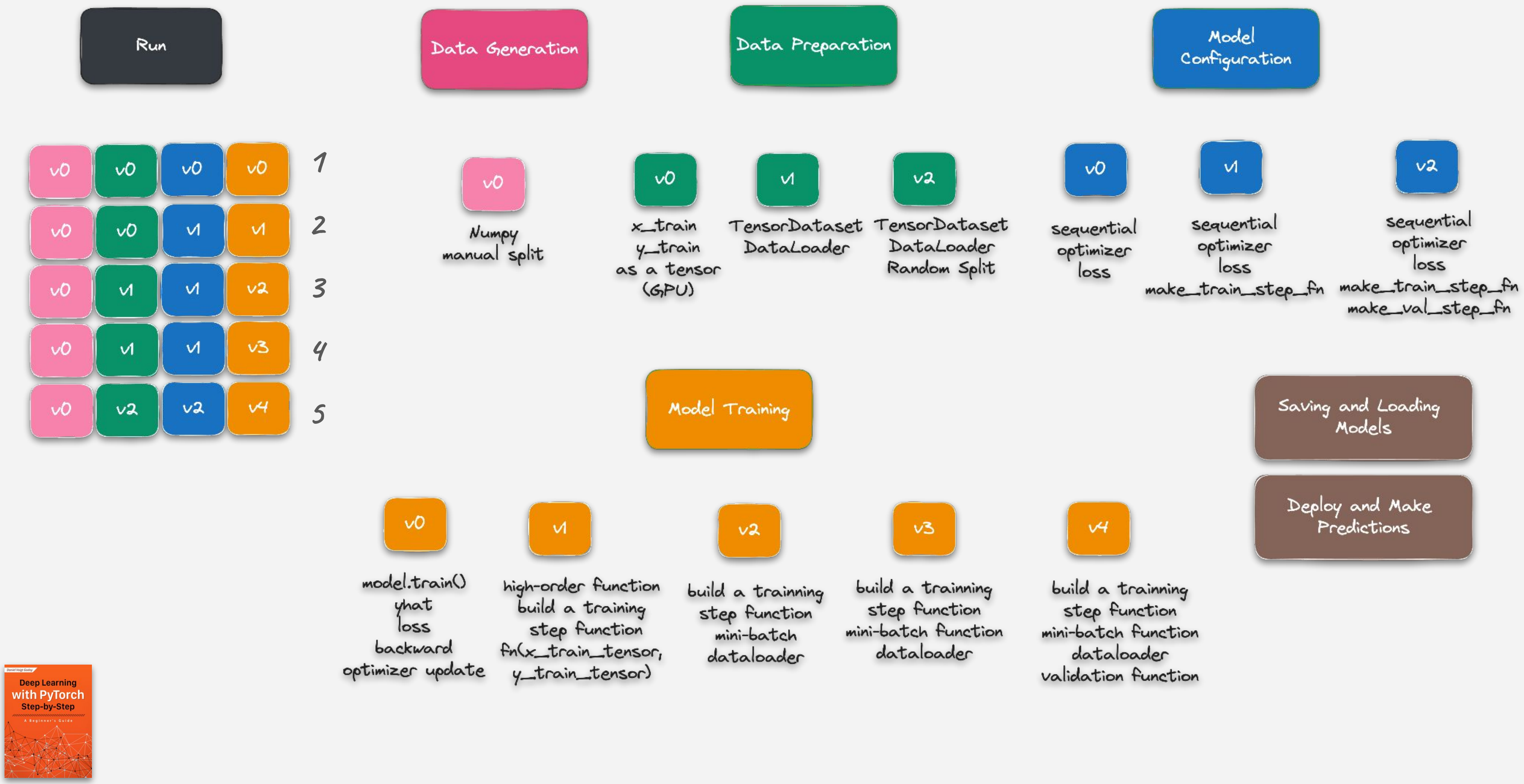
Part 01

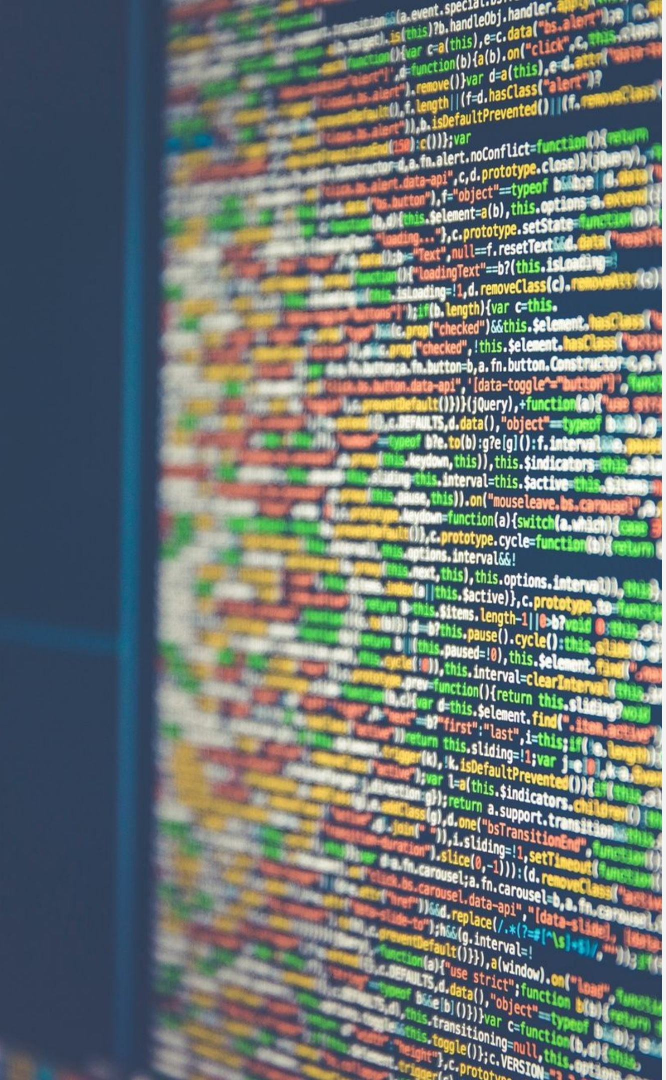
Ivanovitch Silva

ivanovitch.silva@ufrn.br

$$y = b + wx + \epsilon$$




Run
Data Generation
Data Preparation
Model Configuration
v0 v0 v0 v0 1
v0 v0 v1 v1 2
v0 v1 v1 v2 3
v0 v1 v1 v3 4
v0 v2 v2 v4 5
v0
Numpy
manual split
v0
x_train
y_train
as a tensor
(GPU)
v1
TensorDataset
DataLoader
v2
TensorDataset
DataLoader
Random Split
v0
sequential
optimizer
loss
v1
sequential
optimizer
loss
make_train_step_fn
v2
sequential
optimizer
loss
make_train_step_fn
make_val_step_fn
Model Training
Saving and Loading Models
Deploy and Make Predictions
v0
model.train()
yhat
loss
backward
optimizer update
v1
high-order function
build a training
step function
fn(x_train_tensor,
y_train_tensor)
v2
build a trainning
step function
mini-batch
dataloader
v3
build a trainning
step function
mini-batch function
dataloader
v4
build a trainning
step function
mini-batch function
dataloader
validation function
Deep Learning
with PyTorch
Step-by-Step

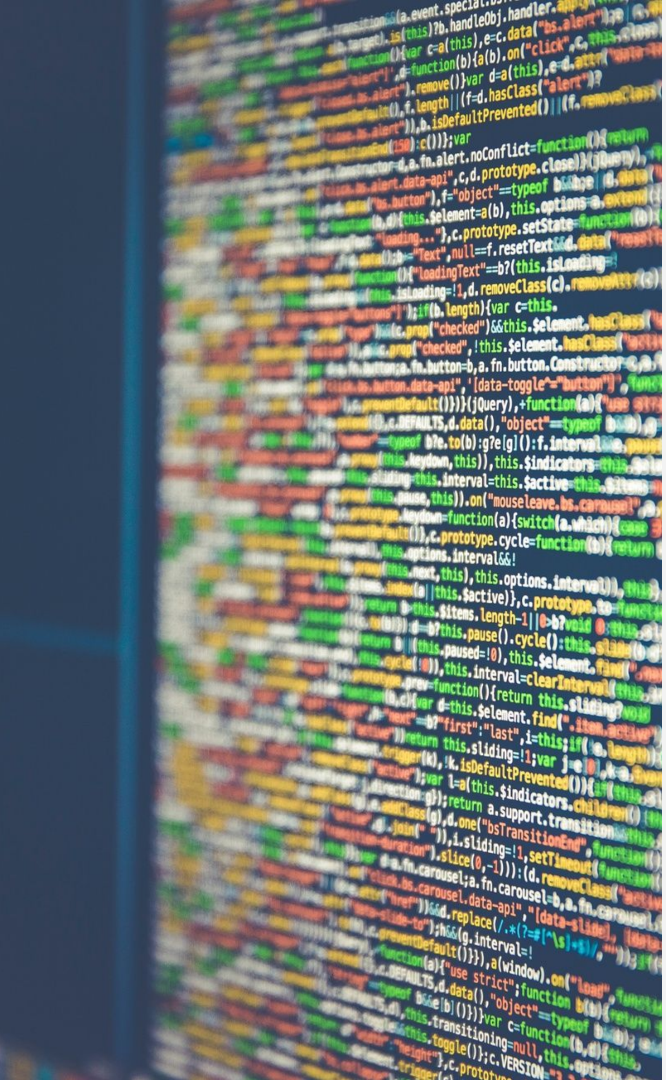


High-Order Functions

Higher-order functions in Python are functions that either **take other functions** as arguments or **return functions** as their results. This concept is key in functional programming and enhances flexibility and expressivity in your code.

Functions Accepting Functions as Arguments

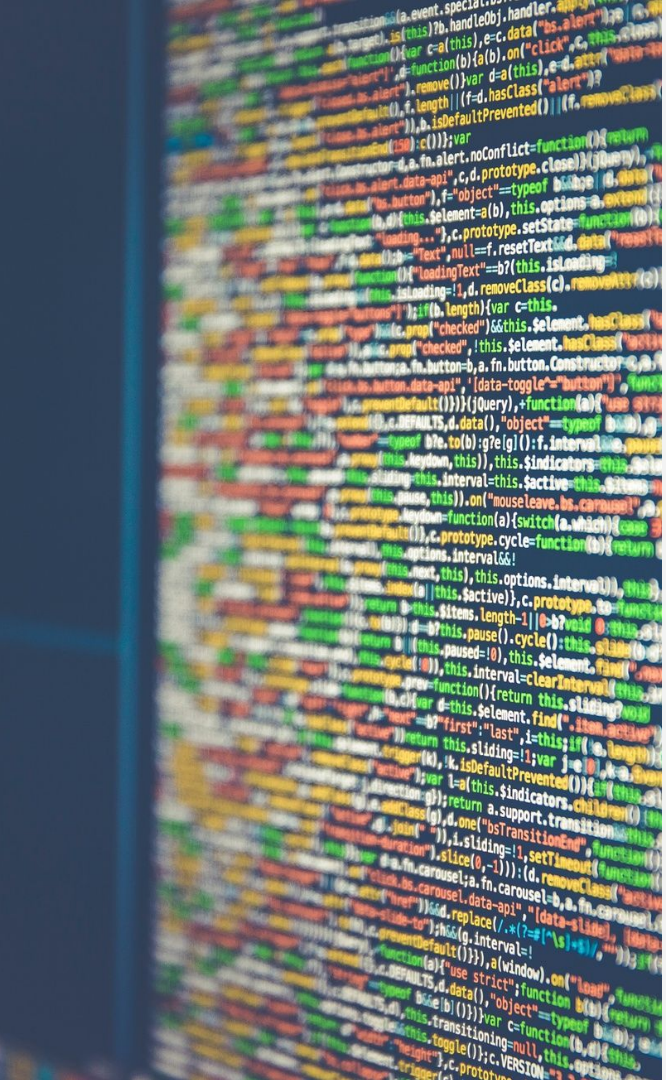
```python
def square(x):
    return x * x

numbers = [1, 2, 3, 4, 5]
squares = map(square, numbers)
print(list(squares))  # Output: [1, 4, 9, 16, 25]
```



High-Order Functions

Functions Returning Functions

```python
def multiplier(n):
    def inner(x):
        return x * n
    return inner

double = multiplier(2)
triple = multiplier(3)

print(double(5))  # Output: 10
print(triple(5))  # Output: 15
```



Why Use High-Order Functions?

- **Code Reusability:** Using higher-order functions, you can write more generic and reusable code. For instance, you can write a general-purpose filter function that can be applied with different criterion functions.
- **Functional Programming:** Higher-order functions are a key component of functional programming, a paradigm emphasizing immutability and the use of functions to transform data.
- **Readability and Expressiveness:** They allow developers to express complex ideas more clearly and concisely.

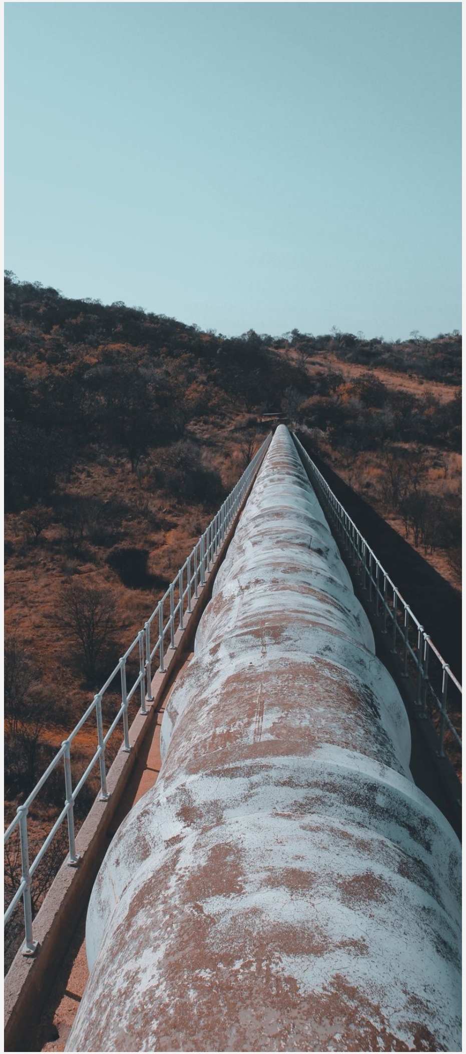

Model lives
on GPU
HYPERX

Dataset

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

class CustomDataset(Dataset):
    def __init__(self, x_tensor, y_tensor):
        self.x = x_tensor
        self.y = y_tensor

    def __getitem__(self, index):
        return (self.x[index], self.y[index])

    def __len__(self):
        return len(self.x)

# Wait, is this a CPU tensor now? Why? Where is .to(device)?
x_train_tensor = torch.from_numpy(x_train).float()
y_train_tensor = torch.from_numpy(y_train).float()

train_data = CustomDataset(x_train_tensor, y_train_tensor)
print(train_data[0])
```

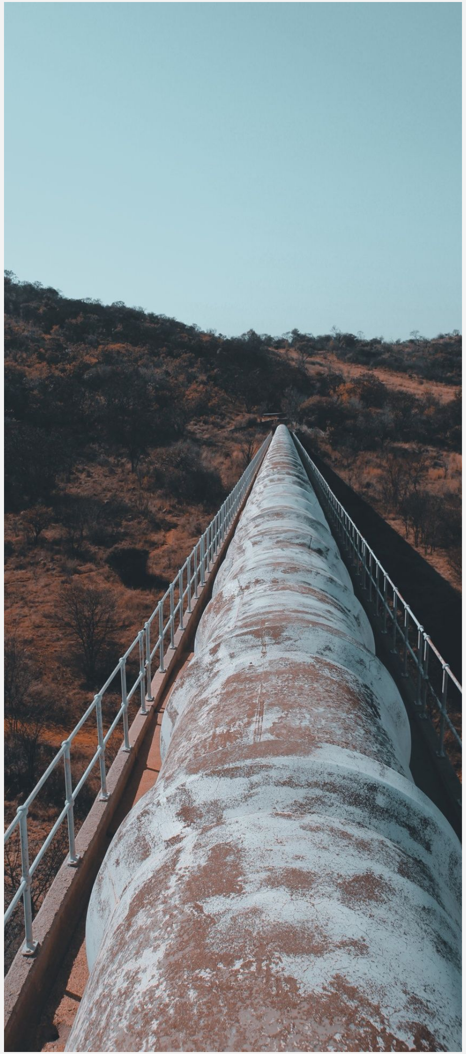
TensorDataset & DataLoader

```
# Our data was in Numpy arrays, but we need to transform them into
# PyTorch's Tensors
x_train_tensor = torch.from_numpy(x_train).float()
y_train_tensor = torch.from_numpy(y_train).float()

# Builds Dataset
train_data = TensorDataset(x_train_tensor, y_train_tensor)

# Builds DataLoader
train_loader = DataLoader(dataset=train_data,
                          batch_size=16,
                          shuffle=True)
```



TensorDataset & DataLoader

```python
# Builds tensors from numpy arrays BEFORE split
x_tensor = torch.from_numpy(x).float()
y_tensor = torch.from_numpy(y).float()

# Builds dataset containing ALL data points
dataset = TensorDataset(x_tensor, y_tensor)

# Performs the split
ratio = .8
n_total = len(dataset)
n_train = int(n_total * ratio)
n_val = n_total - n_train

train_data, val_data = random_split(dataset, [n_train, n_val])

# Builds a loader of each set
train_loader = DataLoader(dataset=train_data,
                          batch_size=16, shuffle=True)
val_loader = DataLoader(dataset=val_data, batch_size=16)
```



Model Configuration

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

torch.manual_seed(42)
# Now we can create a model and send it at once to the device
model = nn.Sequential(nn.Linear(1, 1)).to(device)

# Defines a SGD optimizer to update the parameters (now retrieved directly from the model)
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

# Creates the train_step function for our model, loss function and optimizer
train_step_fn = make_train_step_fn(model, loss_fn, optimizer)

# Creates the val_step function for our model and loss function
val_step_fn = make_val_step_fn(model, loss_fn)
```

Model Configuration

```
def make_train_step_fn(model, loss_fn, optimizer):
    # Builds function that performs a step in the train loop
    def perform_train_step_fn(x, y):
        # Sets model to TRAIN mode
        model.train()

        # Step 1 - Computes our model's predicted output - forward pass
        yhat = model(x)
        # Step 2 - Computes the loss
        loss = loss_fn(yhat, y)
        # Step 3 - Computes gradients for both "a" and "b" parameters
        loss.backward()
        # Step 4 - Updates parameters using gradients and the learning rate
        optimizer.step()
        optimizer.zero_grad()

        # Returns the loss
        return loss.item()

    # Returns the function that will be called inside the train loop
    return perform_train_step_fn
```

Model Configuration

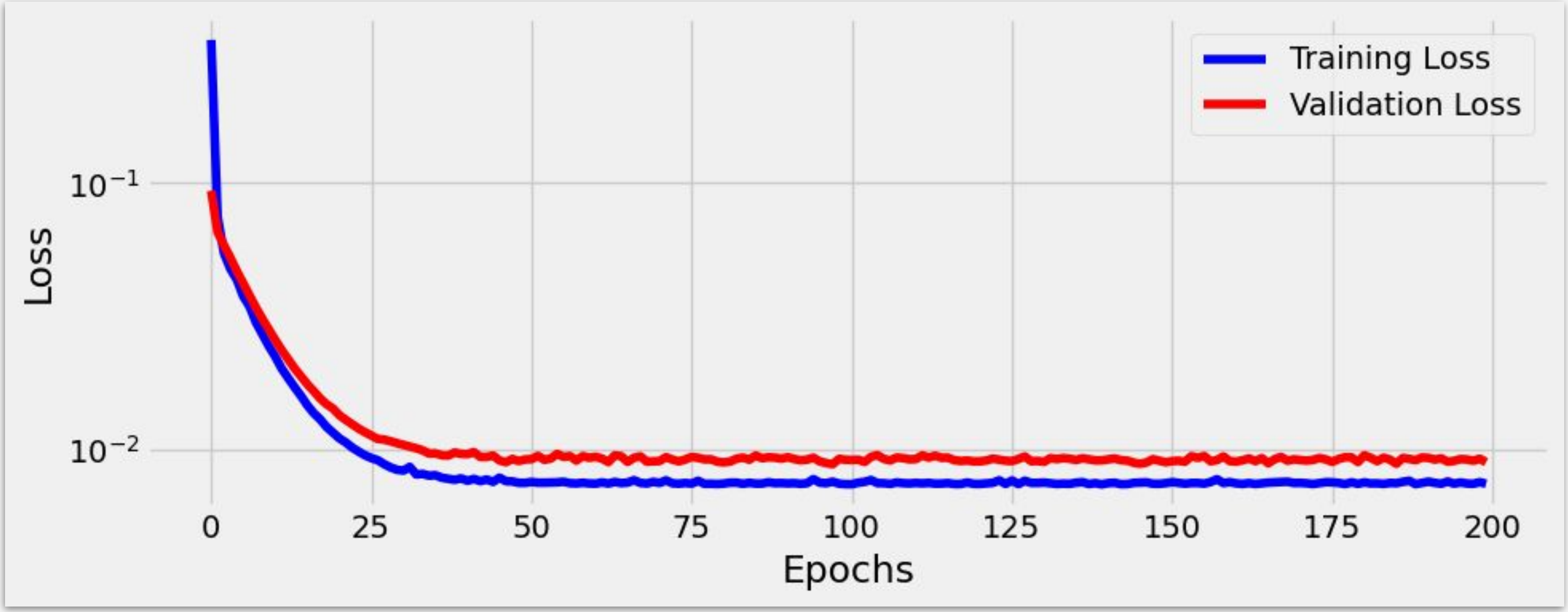
```python
def make_val_step_fn(model, loss_fn):
   # Builds function that performs a step in the validation loop
   def perform_val_step_fn(x, y):
       # Sets model to EVAL mode
       model.eval()

       # Step 1 - Computes our model's predicted output - forward pass
       yhat = model(x)
       # Step 2 - Computes the loss
       loss = loss_fn(yhat, y)
       # There is no need to compute Steps 3 and 4, since we don't update
parameters during evaluation
       return loss.item()

   return perform_val_step_fn
```

Training

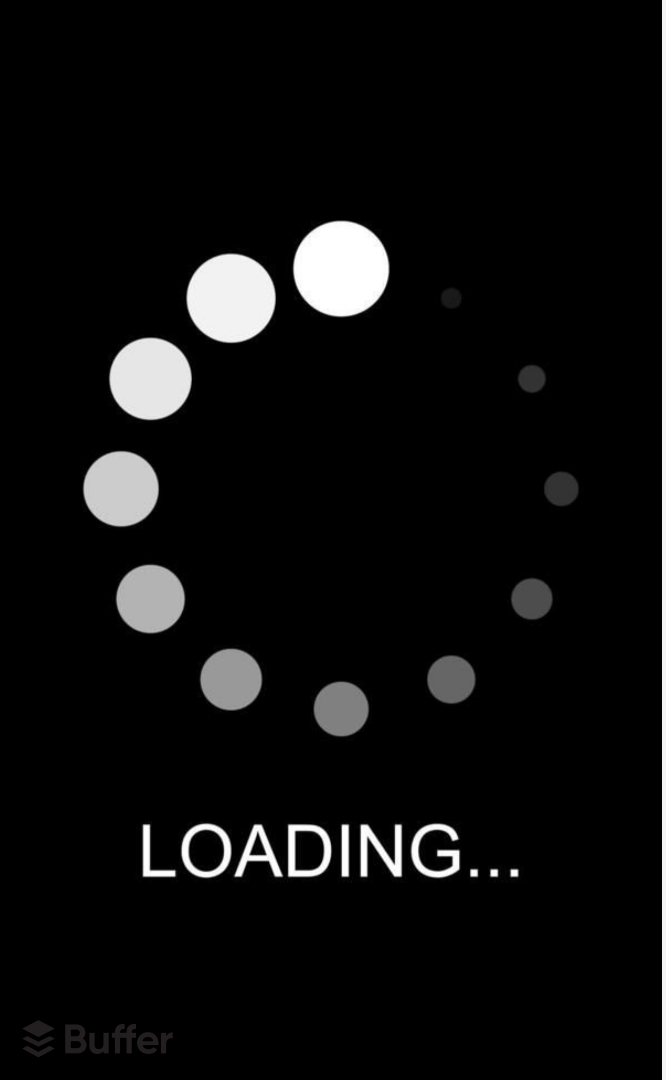
```python
# Defines number of epochs
n_epochs = 200

losses = []
val_losses = []

for epoch in range(n_epochs):
    # inner loop
    loss = mini_batch(device, train_loader, train_step_fn)
    losses.append(loss)

    # VALIDATION
    # no gradients in validation!
    with torch.no_grad():
        val_loss = mini_batch(device, val_loader, val_step_fn)
        val_losses.append(val_loss)
```

Training

```
def mini_batch(device, data_loader, step_fn):
   mini_batch_losses = []
   for x_batch, y_batch in data_loader:
       x_batch = x_batch.to(device)
       y_batch = y_batch.to(device)

       mini_batch_loss = step_fn(x_batch, y_batch)
       mini_batch_losses.append(mini_batch_loss)

   loss = np.mean(mini_batch_losses)
   return loss
```



Training Loss
Validation Loss
10^{-1}
10^{-2}
Loss
0
25
50
75
100
125
150
175
200
Epochs



Saving and Loading Models

```
checkpoint = {'epoch': n_epochs,
              'model_state_dict': model.state_dict(),
              'optimizer_state_dict': optimizer.state_dict(),
              'loss': losses,
              'val_loss': val_losses}

torch.save(checkpoint, 'model_checkpoint.pth')
```

```
checkpoint = torch.load('model_checkpoint.pth',weights_only=False)

model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])

saved_epoch = checkpoint['epoch']
saved_losses = checkpoint['loss']
saved_val_losses = checkpoint['val_loss']

model.train() # always use TRAIN for resuming training
```

Make Predictions

```
checkpoint = torch.load('model_checkpoint.pth',weights_only=False)

model.load_state_dict(checkpoint['model_state_dict'])

new_inputs = torch.tensor([[.20], [.34], [.57]])

model.eval() # always use EVAL for fully trained models!
model(new_inputs.to(device))

tensor([[1.4174],
        [1.6896],
        [2.1366]], device='cuda:0')
```

Going Classy

Object-Oriented Programming - OOP

week04a.ipynb, week04b.ipynb