PyTorch
Hyperparameter Tuning and Best Practices

Lecture 13 for 14-763/18-763

Guannan Qu

Oct 21, 2024

Recall: Stochastic Gradient Descent

One epoch means go through all batches one time.

Use the 1st batch to do a GD step

Use the 2nd batch to do a GD step

…

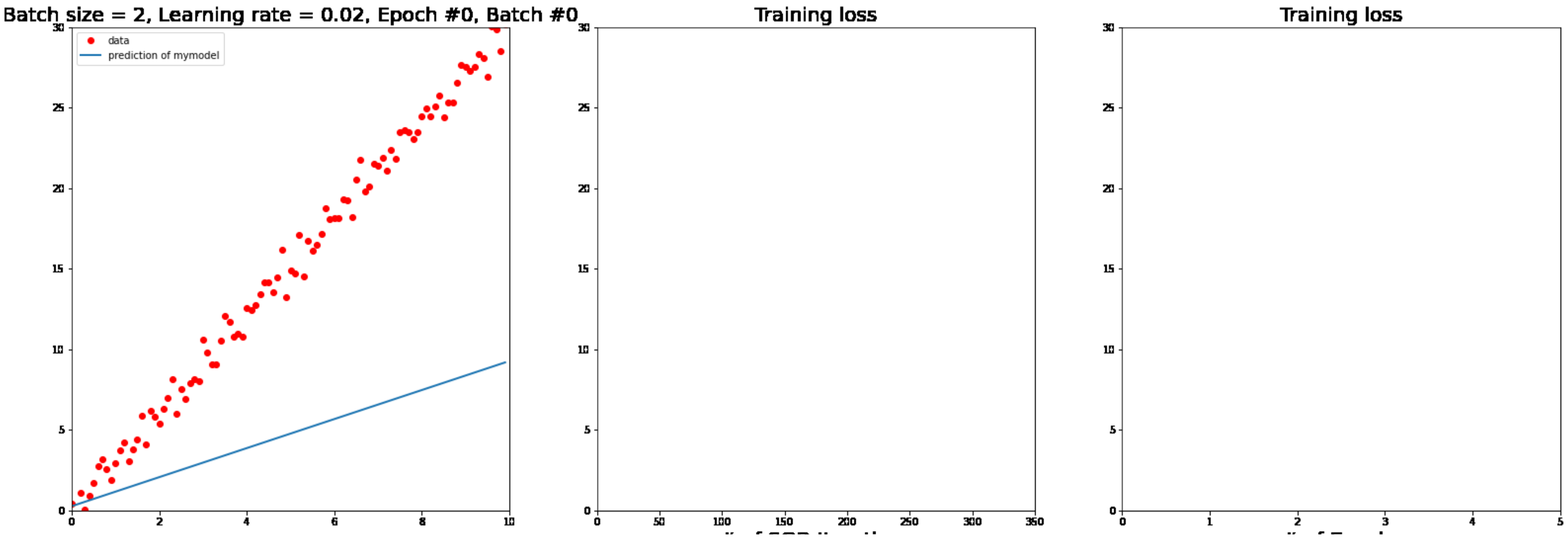
…

Batch size = 2

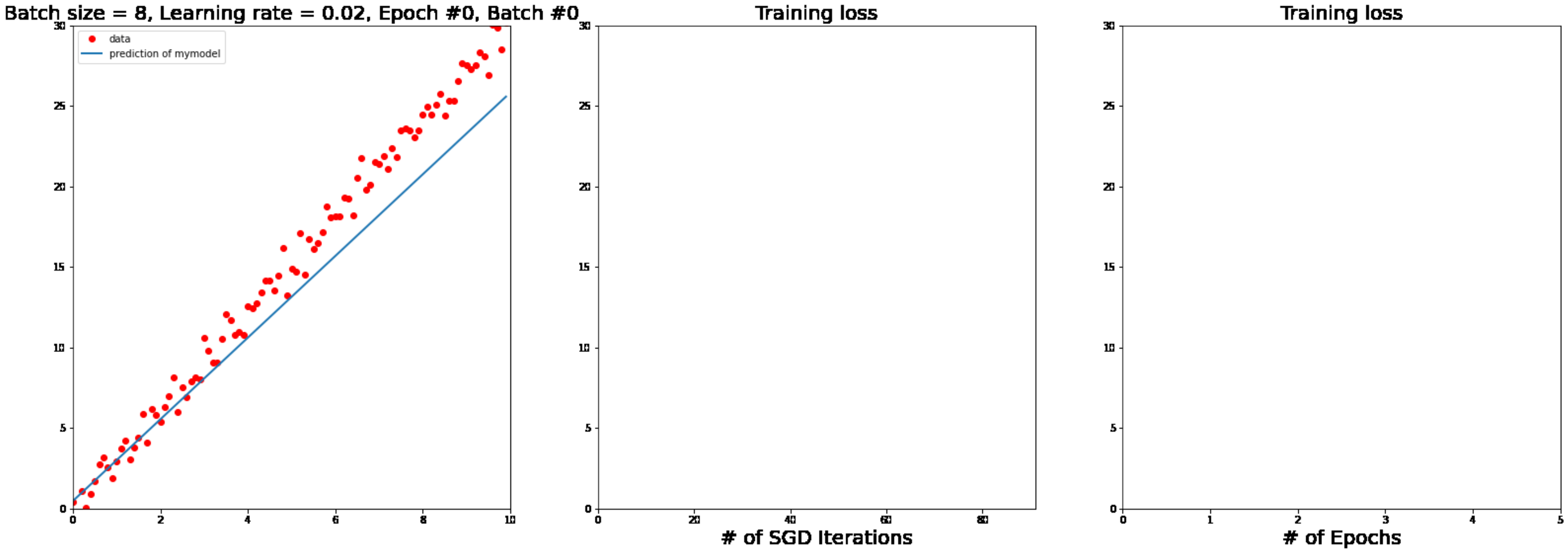
Batched Dataset

Elements
Element 1-2
Element 3-4
Element 5-6
Element 7-8
…

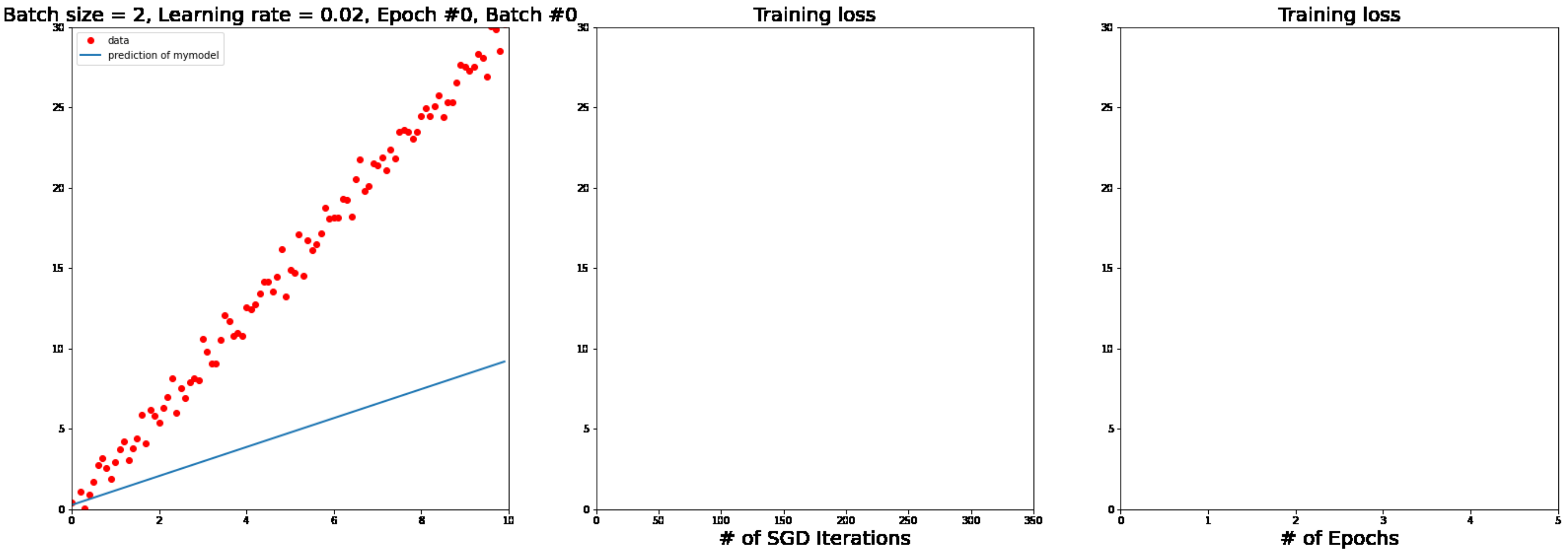
Batch_size=2
Learning rate =0.02



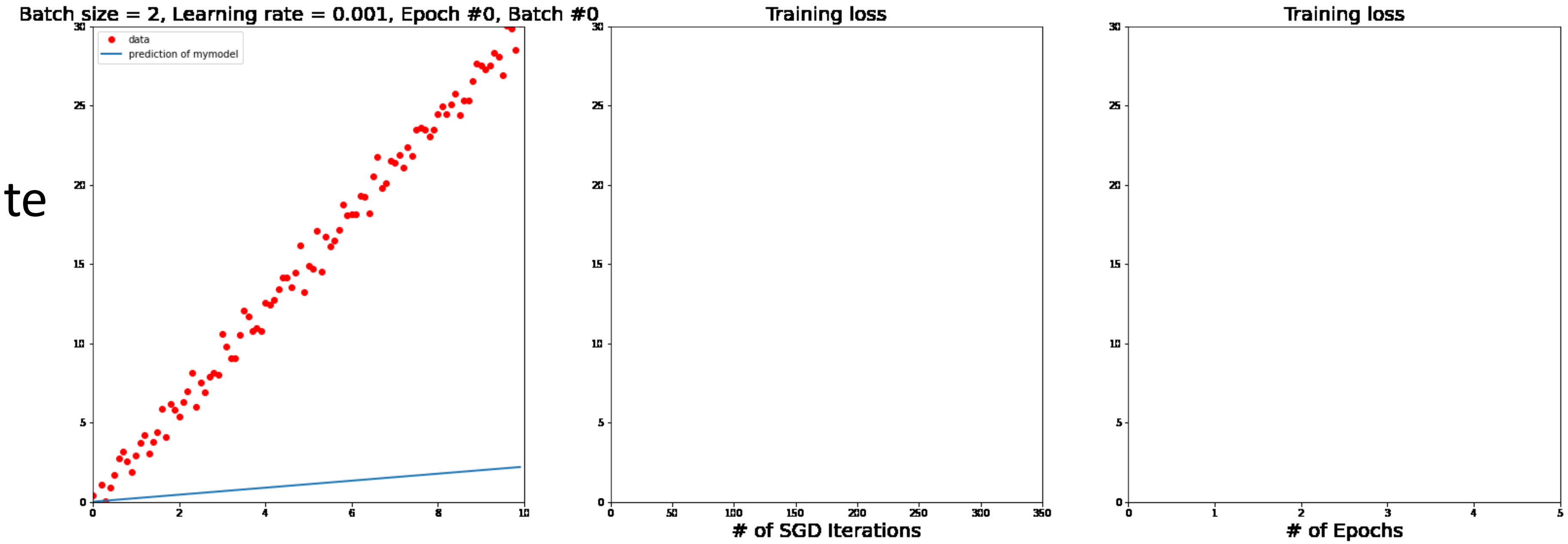
Increase batch_size to 8



Batch_size=2
Learning rate =0.02



Decrease learning rate to 0.001



Recall: Build Neural Network in PyTorch

```
class myMultiLayerPerceptron(nn.Module):
    def __init__(self,input_dim,output_dim):
        super().__init__()
        self.sequential = nn.Sequential(  # here we stack multiple layers together
            nn.Linear(input_dim,20),
            nn.ReLU(),
            nn.Linear(20,20),
            nn.ReLU(),
            nn.Linear(20,20),
            nn.ReLU(),
            nn.Linear(20,20),
            nn.ReLU(),
            nn.Linear(20,output_dim)
        )
    def forward(self,x):
        y = self.sequential(x)
        return y
```

Overall, we create a “Sequential” of layers

The first layer with width 20

The second layer with width 20

The third layer with width 20

The fourth layer with width 20

The output layer

Recall: Training Loops

mymodel is now the neural network we just defined

```
mymodel = myMultiLayerPerceptron(1,1) # creating a model instance with input dimension 1
```

```
# Three hyper parameters for training
lr = .04
batch_size = 10
N_epochs = 160
```

Three hyper parameters

```
# Create dataloaders for training and validation
train_dataloader = DataLoader(train_dataset, batch_size = batch_size, shuffle = True)
validate_dataloader = DataLoader(validate_dataset,batch_size = batch_size,shuffle = True)

# Create optimizer
optimizer = torch.optim.SGD(mymodel.parameters(), lr = lr) # this line creates a optimizer,
```

DataLoader and Optimizer

Recall: Training Loops

The training loop is identical as before!

```
for epoch in range(N_epochs):
    batch_loss = []
    for batch_id, (x_batch, y_batch) in enumerate(train_dataloader):
        gd_steps+=1
        # pass input data to get the prediction outputs by the current model
        prediction = mymodel(x_batch)

        # compare prediction and the actual output and compute the loss
        loss = torch.mean((prediction - y_batch)**2)

        # compute the gradient
        optimizer.zero_grad()
        loss.backward()

        # update parameters
        optimizer.step()
```

Forward pass

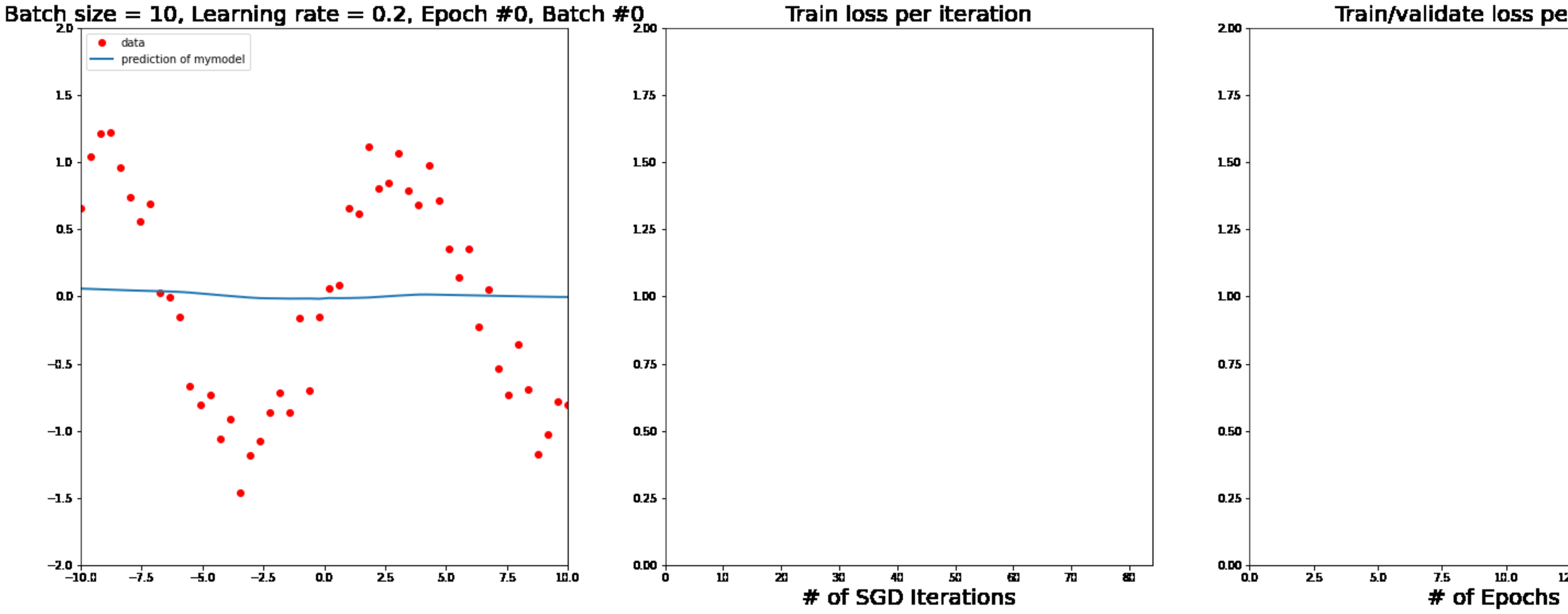
Backward pass and compute gradient.

Run a gradient descent step

Visualizing Neural Network Training

Let’s now visualize the training process and tune hyperparameters!

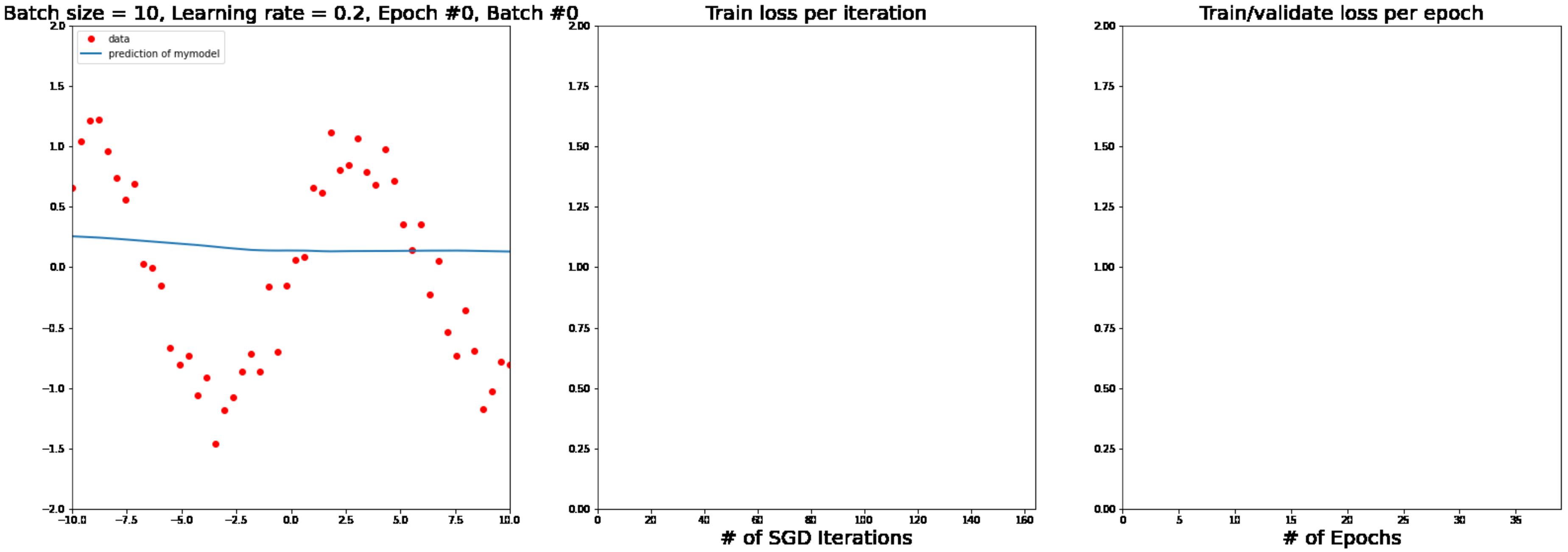
Learning rate = 0.2, N_epochs = 20, batch size = 10



Visualizing Neural Network Training

Let's try a bit more epochs!

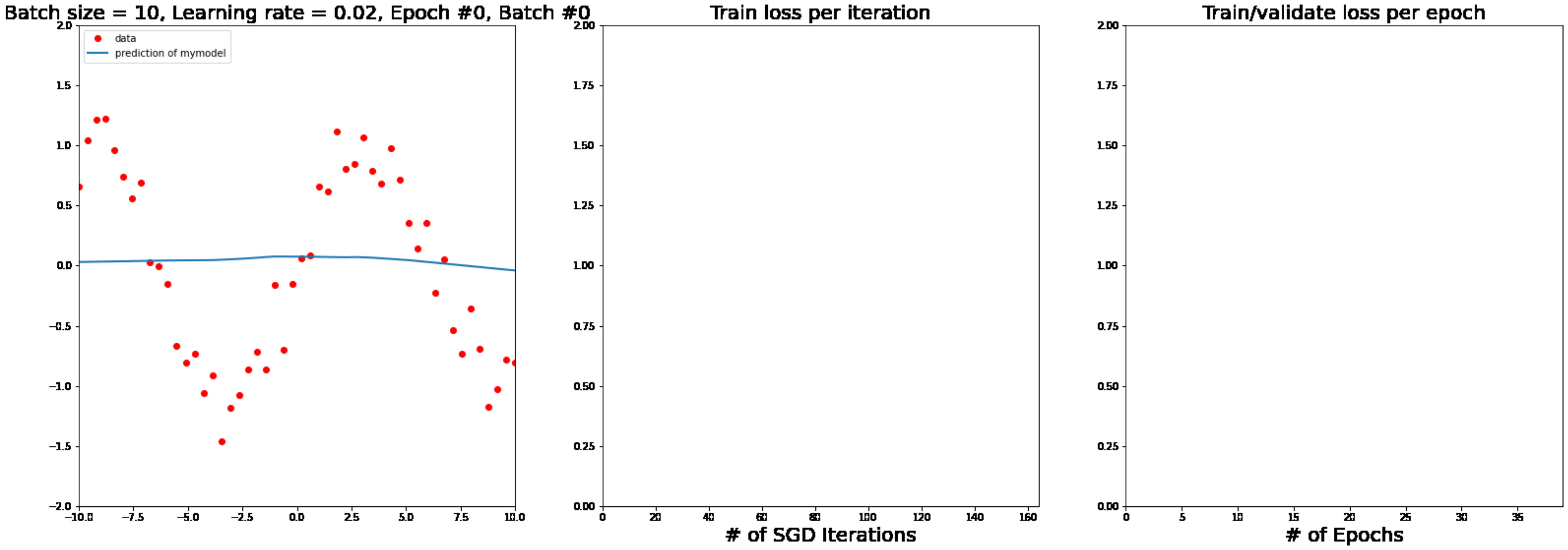
Learning rate = 0.2, N_epochs = 40, batch size = 10



Visualizing Neural Network Training

Let’s lower the learning rate!

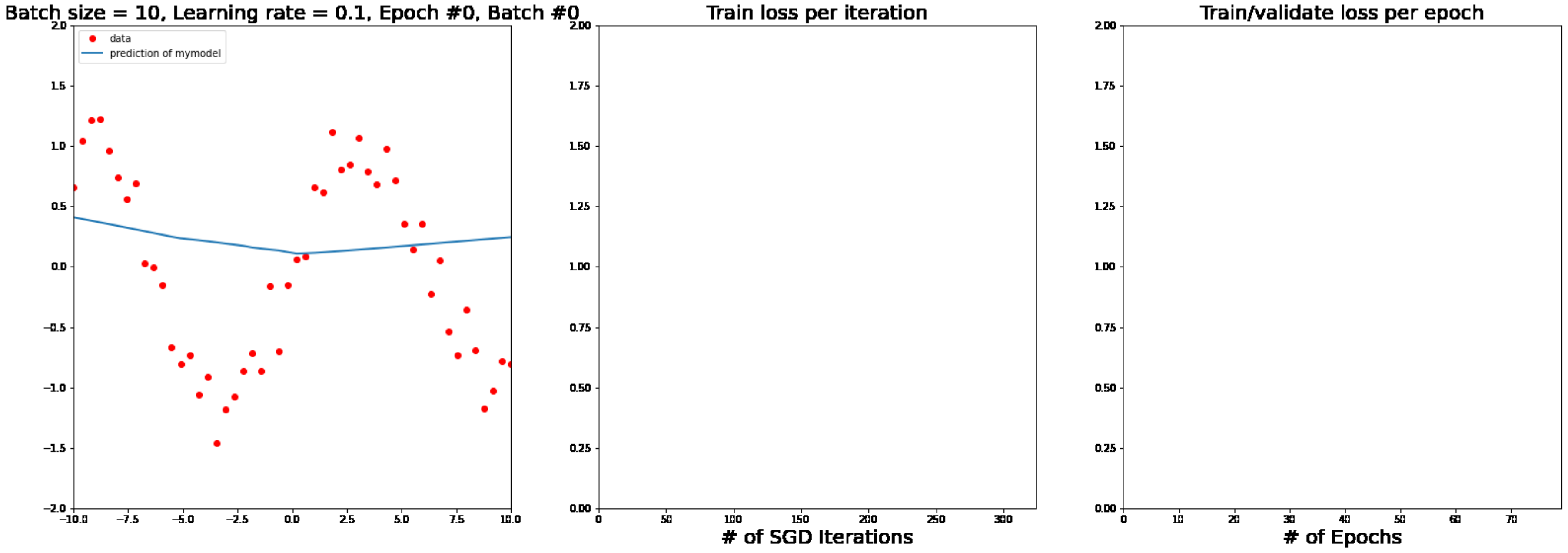
Learning rate = 0.02, N_epochs = 40, batch size = 10



Visualizing Neural Network Training

Let’s increase the learning rate and increase N_epochs!

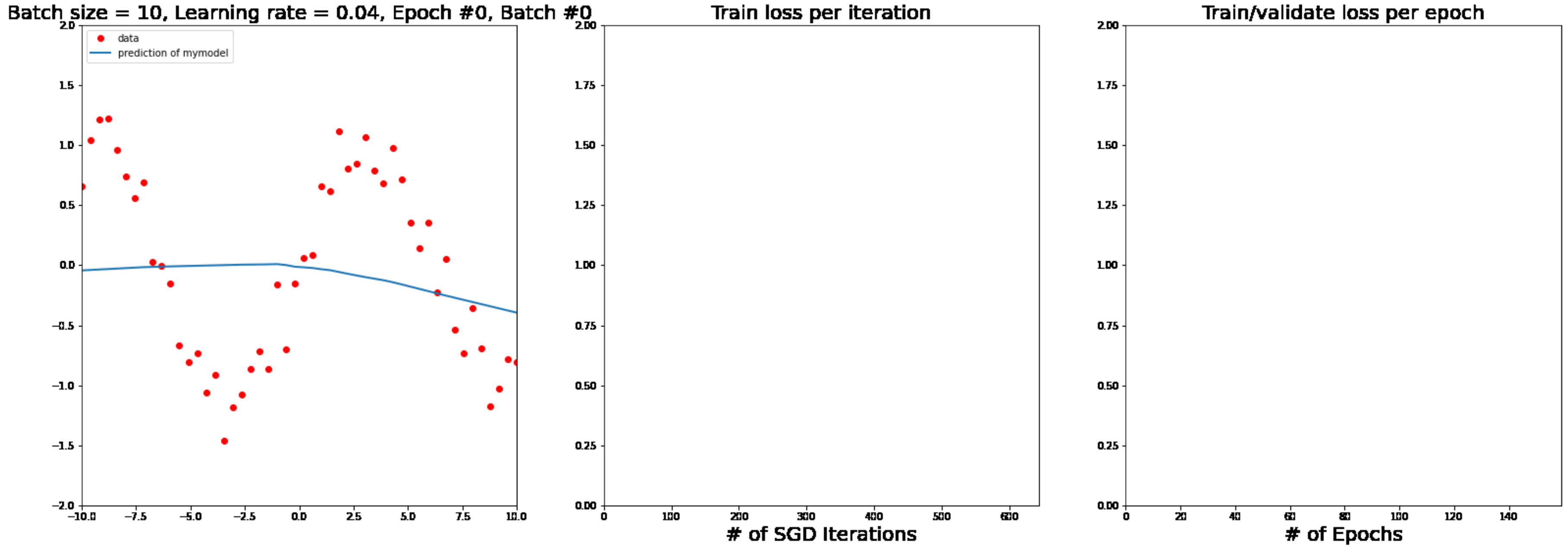
Learning rate = 0.1, N_epochs = 80, batch size = 10



Visualizing Neural Network Training

Let's lower the learning rate slightly and increase N_epochs!

Learning rate = 0.04, N_epochs = 160, batch size = 10



Lessons Learned on Hyperparameter Tuning

- Too much randomness and oscillations? Loss not improving?
  - Reduce learning rate
  - Increase batch size (this is less common)
- Converging too slow?
  - Increase learning rate
- The loss seems to decreasing and the model is learning something, but it has not fully converged at the end of the training?
  - Increase the number of epochs

More on Tuning for Neural Networks

Two groups of parameters

- Training parameters: learning rate, batzh_size, N_epochs
- Model parameters: depth, width, etc

Typical tuning strategy

- Figure out training parameters that can reliably work well regardless of model parameters
    - This often means being conservative and choose a small learning rate and large number of epochs
    - Fixing the training parameters, try different model parameters

Summary So Far

Build Models

- Subclassing nn.module
- Using nn.Sequential() to connect nn.Linear and nn.ReLU together
- Define forward function

Create Dataset

- Subclassing torch.utils.data.Dataset
- Define __len__() and __getitem__() function

Training Loops

- Create Optimizer, DataLoader
- Nested for-loop, outer-loop for epochs, inner-loop for batches
- Forward, zero_grad, backward, step
- Helpful to record train/validate loss (and other metrics) for each epoch

Up next

- Additional tools
  - Adam optimizer, learning rate scheduler
  - Other built-in activations, loss functions
- NSL-KDD Example: best practices in PyTorch

Optimizers in PyTorch

Adam is another very popular optimizer!

We have been using torch.optim.SGD

Adadelta	Implements Adadelta algorithm.
Adagrad	Implements Adagrad algorithm.
Adam	Implements Adam algorithm.
AdamW	Implements AdamW algorithm.
SparseAdam	Implements lazy version of Adam algorithm suitable for sparse tensors.
Adamax	Implements Adamax algorithm (a varia of Adam based on infinity norm).
ASGD	Implements Averaged Stochastic Gradient Descent.
LBFGS	Implements L-BFGS algorithm, heavily inspired by minFunc.
NAdam	Implements NAdam algorithm.
RAdam	Implements RAdam algorithm.
RMSprop	Implements RMSprop algorithm.
Rprop	Implements the resilient backpropagation algorithm.
SGD	Implements stochastic gradient descent (optionally with momentum).

What is Adam?

Published as a conference paper at ICLR 2015

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Diederik P. Kingma*
University of Amsterdam, OpenAI
dpkingma@openai.com

Jimmy Lei Ba*
University of Toronto
jimmy@psi.utoronto.ca

Key ingredient (beyond the scope of this course):

- SGD with momentum
- Rescaled gradients

Very popular, with paper cited more than 190k times

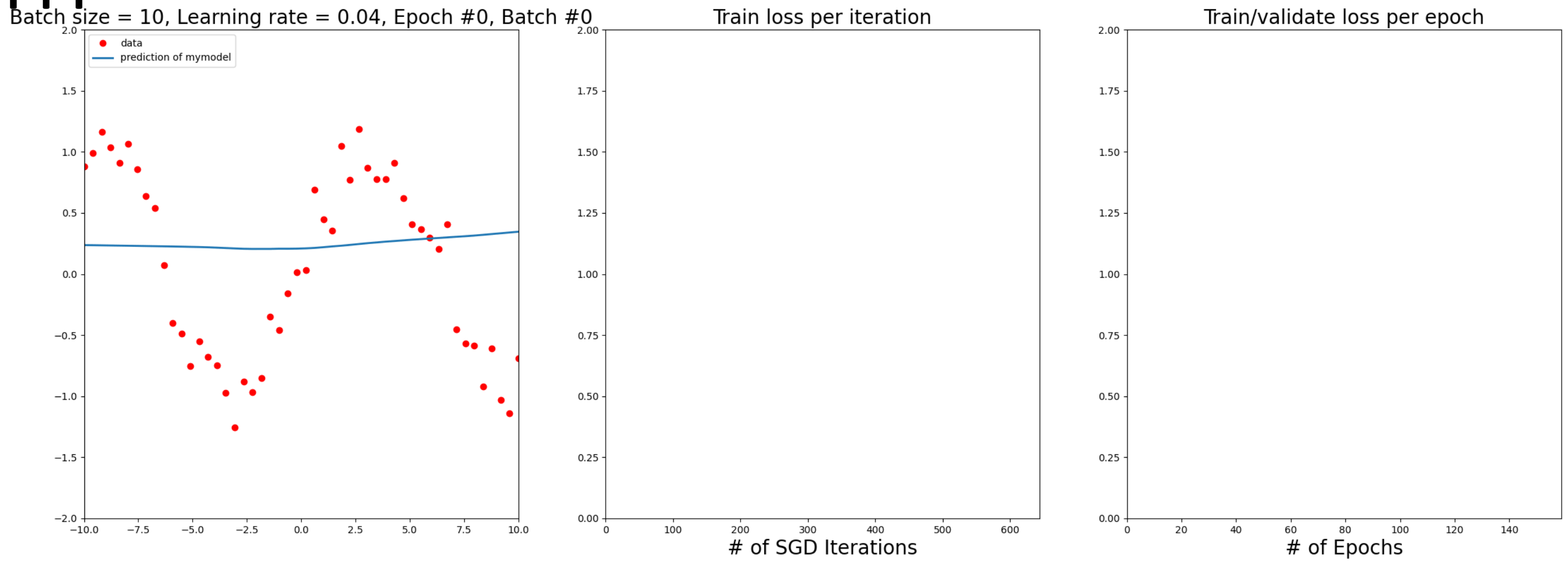
Implement Adam in PyTorch

```
# Create optimizer - choose between SGD or Adam
# optimizer = torch.optim.SGD(mymodel.parameters(), lr = lr)
optimizer = torch.optim.Adam(mymodel.parameters(), lr = lr)
```

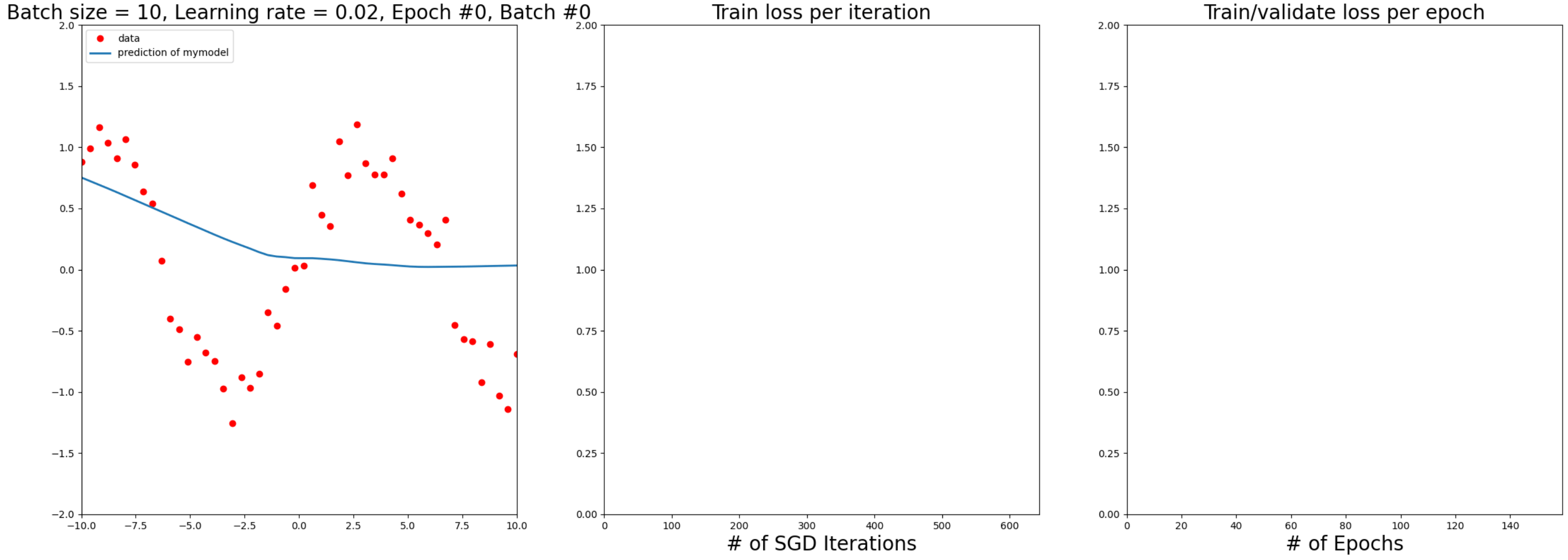
Just use optim.Adam instead of optim.SGD! The rest is identical

SGD vs Adam

SGD Optimizer
Batch size = 10
Learning rate = 0.04
N_epochs = 160
(The parameters we got in last lecture)



Adam Optimizer
Batch size = 10
Learning rate = 0.02
N_epochs = 160
(The parameters we got in last lecture)



Learning Rate Scheduler

- Recall in the tuning example, we encounter situations where initially the learning is fast, but towards the end the learning is "unstable"
- We had to decrease the learning rate, but a better solution is to use a large learning rate at the beginning, and a smaller learning rate at the end.
- torch.optim.lr_scheduler provides methods to adjust learning rate
- For example, ExponentialLR shrinks the learning rate by a fixed ratio at each epoch

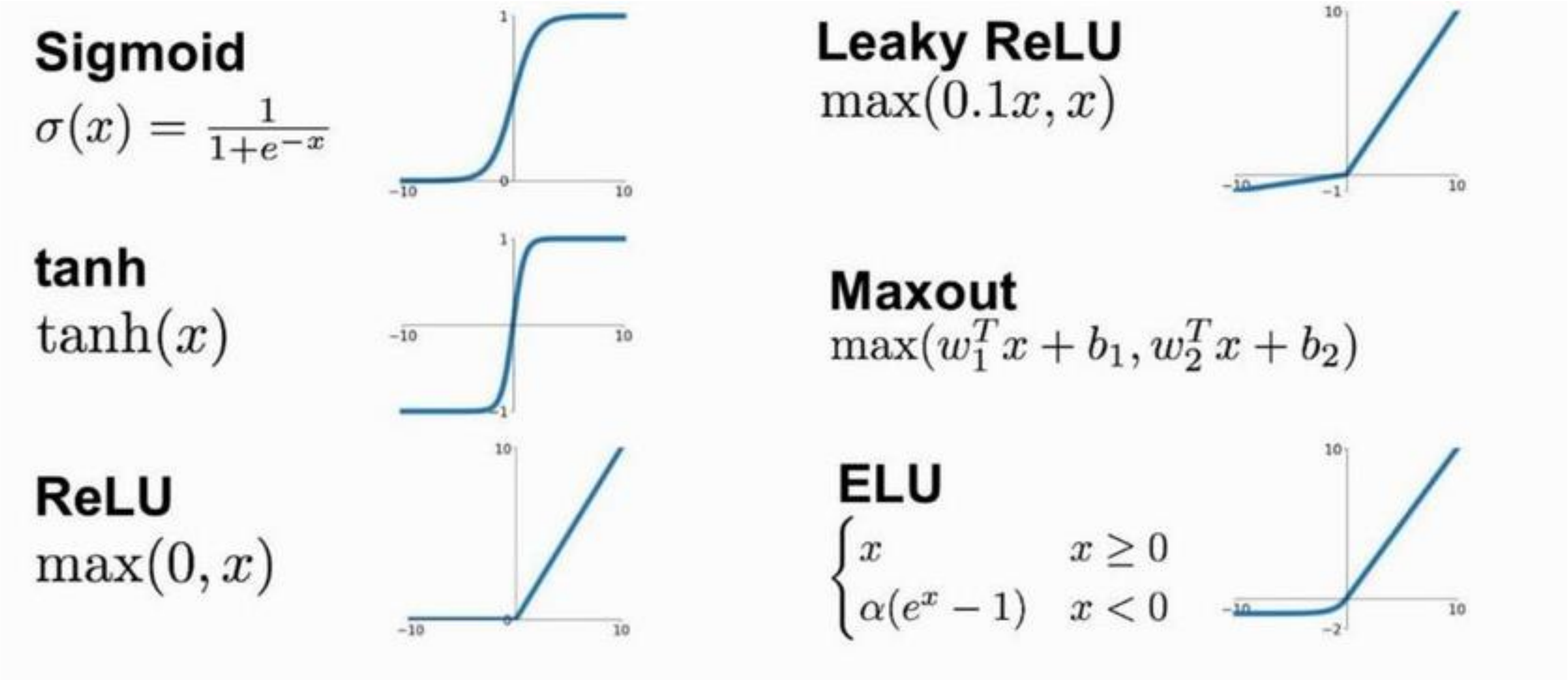
```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
scheduler = ExponentialLR(optimizer, gamma=0.9)

for epoch in range(20):
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
    scheduler.step()
```

Built-in Layers in PyTorch

- Containers
- Convolution Layers
- Pooling layers
- Padding Layers
- Non-linear Activations (weighted sum, nonlinearit
- Non-linear Activations (other)
- Normalization Layers
- Recurrent Layers
- Transformer Layers
- Linear Layers
- Dropout Layers
- Sparse Layers
- Distance Functions
- Loss Functions
- Vision Layers
- Shuffle Layers
- DataParallel Layers (multi-GPU, distributed)
- Utilities
- Quantized Functions
- Lazy Modules Initialization

`nn.ReLU`	Applies the rectified linear unit function element-wise:
`nn.ReLU6`	Applies the element-wise function:
`nn.RReLU`	Applies the randomized leaky rectified liner unit function, element-wise, as described in the paper:
`nn.SELU`	Applied element-wise, as:
`nn.CELU`	Applies the element-wise function:
`nn.GELU`	Applies the Gaussian Error Linear Units function:
`nn.Sigmoid`	Applies the element-wise function:

Comparison of Activation Functions



Jayawardana, Rahul & Bandaranayake, Thusitha. (2021). ANALYSIS OF OPTIMIZING NEURAL NETWORKS AND ARTIFICIAL INTELLIGENT MODELS FOR GUIDANCE, CONTROL, AND NAVIGATION SYSTEMS.

Built-in Layers in PyTorch

- Containers
- Convolution Layers
- Pooling layers
- Padding Layers
- Non-linear Activations (weighted sum, nonlinearity)
- Non-linear Activations (other)
- Normalization Layers
- Recurrent Layers
- Transformer Layers
- Linear Layers
- Dropout Layers
- Sparse Layers
- Distance Functions
- Loss Functions
- Vision Layers
- Shuffle Layers
- DataParallel Layers (multi-GPU, distributed)
- Utilities
- Quantized Functions
- Lazy Modules Initialization

`nn.MSELoss`	Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.
`nn.CrossEntropyLoss`	This criterion computes the cross entropy loss between input logits and target.
`nn.KLDivLoss`	The Kullback-Leibler divergence loss.
`nn.BCELoss`	Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities:

Example: NSLKDD

- Convert NSL-KDD into torch.utils.data.Dataset
- Build NN with **Tanh()** activation
- Training loop with **built-in loss functions**
  - Best practices in writing training loops

Convert NSL-KDD into torch.utils.data.Dataset

- Convert the spark data frame to pandas, and then numpy
- Use torch.from_numpy() to convert numpy array to torch tensor

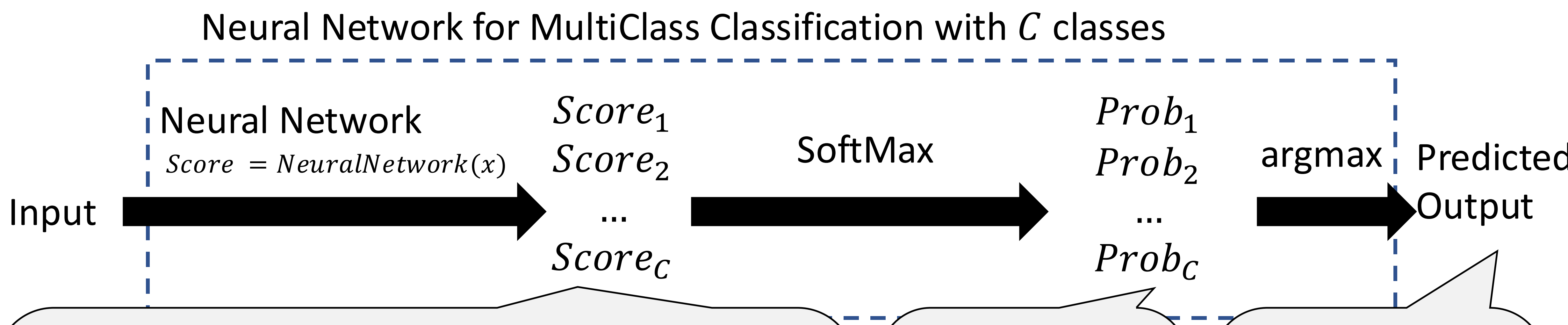
```
x_train = torch.from_numpy(np.array(nslkdd_df_train_pandas['features'].values.tolist(),np.float32))
y_train = torch.from_numpy(np.array(nslkdd_df_train_pandas['outcome'].values.tolist(),np.int64))

x_validate = torch.from_numpy(np.array(nslkdd_df_validate_pandas['features'].values.tolist(),np.float32))
y_validate = torch.from_numpy(np.array(nslkdd_df_validate_pandas['outcome'].values.tolist(),np.int64))

x_test = torch.from_numpy(np.array(nslkdd_df_test_pandas['features'].values.tolist(),np.float32))
y_test = torch.from_numpy(np.array(nslkdd_df_test_pandas['outcome'].values.tolist(),np.int64))
```

Convert NSL-KDD into torch.utils.data.Dataset

- Convert the tensors to datasets

```
class MyDataset(Dataset):
    def __init__(self,x,y):
        self.x = x
        self.y = y

    def __len__(self):
        return self.x.shape[0]

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

train_dataset = MyDataset(x_train,y_train)
validate_dataset = MyDataset(x_validate,y_validate)
test_dataset = MyDataset(x_test,y_test)
```

Neural Network for MultiClass Classification



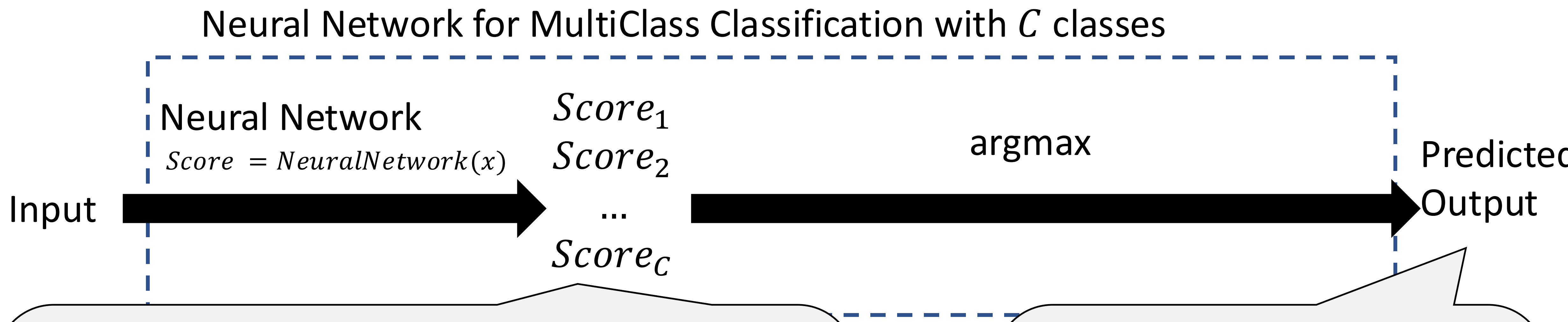
PyTorch convention: The neural network's output should be the score (logit) of each class.

Meaning of score: the larger the score of a class, the more likely the class is the true output.

The predicted probability of each class

The predicted output is the class that has the **highest probability (the argmax)**

Neural Network for MultiClass Classification



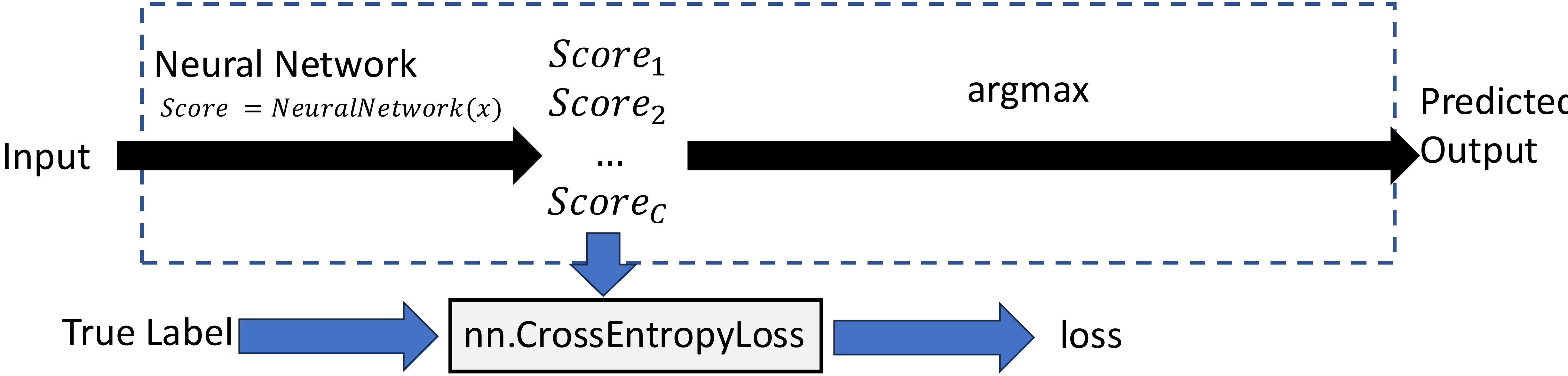
PyTorch convention: The neural network's output should be the score (logit) of each class.

Meaning of score: the larger the score of a class, the more likely the class is the true output.

Equivalently, the predicted output is the class that has the **highest score**
(the argmax of the scores)

Neural Network for MultiClass Classification

Neural Network for MultiClass Classification with C classes



Note: we don't use ``accuracy'' as the loss as accuracy is not differentiable. CrossEntropy is the standard loss for classification problems

torch.nn.functional.cross_entropy(*input*, *target*, *weight=None*, *size_average=None*, *ignore_index=-100*, *reduce=None*, *reduction='mean'*, *label_smoothing=0.0*) [SOURCE]

This criterion computes the cross entropy loss between input logits and target.

See `CrossEntropyLoss` for details.

Parameters

- **input** (*Tensor*) – Predicted unnormalized logits; see Shape section below for

Build NN with **Tanh()** activation

Suppose input dimension of the NSL-KDD problem is 113, and the output is a label that has two classes (0 or 1). How should we set the output of the neural network?

Build NN with **Tanh()** activation

```
class myMultiLayerPerceptron_TahnActivation(nn.Module):
    def __init__(self,input_dim,output_dim):
        super().__init__()
        self.sequential = nn.Sequential(  # here we stack
            nn.Linear(input_dim,20),
            nn.Tanh(),
            nn.Linear(20,20),
            nn.Tanh(),
            nn.Linear(20,20),
            nn.Tanh(),
            nn.Linear(20,20),
            nn.Tanh(),
            nn.Linear(20,output_dim)
        )
    def forward(self,x):
        y = self.sequential(x)
        return y
```

Overall, we create a “Sequential” of layers

The first layer with width 20 and Tanh() activation!

The second layer with Tanh() activation!

The third layer with Tanh() activation!

The fourth layer with Tanh() activation!

The forward method just uses the sequential we created to compute the output.

Training Loops: Before the Loop Starts

Create Model

```
mymodel = myMultiLayerPerceptron_TahnActivation(x_train.shape[1],2) # creating a model instan
```

```
# Three hyper parameters for training
lr = .005
batch_size = 64
N_epochs = 10
```

Define hyper parameters

Good practice: move all hyper parameter definitions to a single place

```
# Create loss function
loss_fun = nn.CrossEntropyLoss()
```

Create loss function.

```
# Create dataloaders for training and validation
train_dataloader = DataLoader(train_dataset, batch_size = batch_size, shuffle = True)
validate_dataloader = DataLoader(validate_dataset,batch_size = batch_size,shuffle = True)

# Create optimizer
optimizer = torch.optim.Adam(mymodel.parameters(), lr = lr) # this line creates a optimizer,
```

Create Train/Validate DataLoader and Optimizer

Good practice: set shuffle = True

Training Loops: Before the Loop Starts

Create lists which we will use to store train/validate loss/accuracy of each epoch

```
losses = [] # training losses of each epoch
accuracies = [] # training accuracies of each epoch

validate_losses = [] # validation losses of each epoch
validate_accuracies = [] # validation accuracies of each epoch

current_best_accuracy = 0.0
```

Keep a variable that will record what is the best validate accuracy seen so far. Initialize this to 0

Training Loops

Create a list that will store the loss/accuracy for each batch

```
for epoch in range(N_epochs):
    # Train loop
    batch_loss = [] # keep a list of losses for different batches in this epoch
    batch_accuracy = [] # keep a list of accuracies for different batches in this epoch
    for x_batch, y_batch in train_dataloader:
        ......
        (Training loop of this epoch)

    validate_batch_loss = [] # keep a list of losses for different validate batches in this
    validate_batch_accuracy = []
    for x_batch, y_batch in validate_dataloader:
        ......
        (Validation loop of this epoch)

    ......
    (End of epoch processing)
```

Let's take a look at this

Good practice:
Do validation at each epoch

Training Loops: Training

```python
batch_loss = [] # keep a list of losses for different batches in this epoch
batch_accuracy = [] # keep a list of accuracies for different batches in this epoch
for x_batch, y_batch in train_dataloader:
    # pass input data to get the prediction outputs by the current model
    prediction_score = mymodel(x_batch)

    # compute the cross entropy loss
    loss = loss_fun(prediction_score,y_batch)

    # compute the gradient
    optimizer.zero_grad()
    loss.backward()

    # update parameters
    optimizer.step()

    # append the loss of this batch to the batch_loss list
    batch_loss.append(loss.detach().numpy())

    # You can also compute other metrics (accuracy) for this batch here
    prediction_label = torch.argmax(prediction_score.detach(),dim=1).numpy()
    batch_accuracy.append( np.sum(prediction_label == y_batch.numpy())/x_batch.shape[0])
```

Forward Pass: make predictions and compute loss
Here we used loss_fun, which was created as
loss_func = nn.CrossEntropyLoss()

Zero-grad and backward pass

Optimizer step

Record the loss of this batch to the batch_loss list

Calculate accuracy of this batch and record it

Training Loops: Training

Details on accuracy calculation:

First need to convert predicted scores to the predicted labels

- The output of NN is prediction_score, which is a tensor of shape (batch_size, 2), i.e. 2 entries for each record in the batch
  - The 1st entry (index 0) is the score for the class label 0 (normal)
  - The 2nd entry (index 1) is the score for the class label 1 (attack)
  - torch.argmax will return the index with the highest score, which is the predicted label

After computing the predicted labels, calculate the fraction of them that are correct

```
# You can also compute other metrics (accuracy) for this batch here
prediction_label = torch.argmax(prediction_score.detach(),dim=1).numpy()
batch_accuracy.append( np.sum(prediction_label == y_batch.numpy())/x_batch.shape[0])
```

Calculate accuracy of this batch and record it

Training Loops: Training

```
for epoch in range(N_epochs):
    # Train loop
    batch_loss = [] # keep a list of losses for different batches in this epoch
    batch_accuracy = [] # keep a list of accuracies for different batches in this epoch
    for x_batch, y_batch in train_dataloader:
        ......
        (Training loop of this epoch)

    validate_batch_loss = [] # keep a list of losses for different validate batches in this
    validate_batch_accuracy = []
    for x_batch, y_batch in validate_dataloader:
        ......
        (Validation loop of this epoch)

    ......
    (End of epoch processing)
```

Let's take a look at this

Training Loops: Validation

```
# Validation loop
validate_batch_loss = [] # keep a list of losses for different validate batches in this epoch
validate_batch_accuracy = []
for x_batch, y_batch in validate_dataloader:
    # pass input data to get the prediction outputs by the current model
    prediction_score = mymodel(x_batch)

    # compare prediction and the actual output and compute the loss
    loss = loss_fun(prediction_score,y_batch)

    # append the lost of this batch to the validate_batch_loss list
    validate_batch_loss.append(loss.detach())

    # You can also compute other metrics (like accuracy) for this batch here
    prediction_label = torch.argmax(prediction_score.detach(),dim=1).numpy()
    validate_batch_accuracy.append( np.sum(prediction_label == y_batch.numpy())/x_batch.shape[0])
```

Forward Pass: make predictions and compute loss

Record the loss of this batch to the batch_loss list

Calculate accuracy of this batch and record it to list validate_batch_loss

Training Loops

```
for epoch in range(N_epochs):
    # Train loop
    batch_loss = [] # keep a list of losses for different batches in this epoch
    batch_accuracy = [] # keep a list of accuracies for different batches in this epoch
    for x_batch, y_batch in train_dataloader:
        ……
        (Training loop of this epoch)

    validate_batch_loss = [] # keep a list of losses for different validate batches in this
    validate_batch_accuracy = []
    for x_batch, y_batch in validate_dataloader:
        ……
        (Validation loop of this epoch)

    ……
    (End of epoch processing)
```

Let’s take a look at this

Training Loops: End of Epoch Processing

```python
# calculate the average train loss and validate loss in this epoch and record them
losses.append(np.mean(np.array(batch_loss)))
validate_losses.append( np.mean(np.array(validate_batch_loss)))
# You can also compute other metrics for this epoch here
accuracies.append(np.mean(np.array(batch_accuracy)))
validate_accuracies.append(np.mean(np.array(validate_batch_accuracy)))

# Printing
print(f"Epoch = {epoch},train_loss={losses[-1]},validate_loss={validate_losses[-1]}")
print(f"Train accuracy = {np.round(accuracies[-1]*100,2)}%, validate accuracy = {np.round(validate_accuracies[-1]*100,2)}% ")

# If the validate metric of this epoch is the best so far, save it
if validate_accuracies[-1]>current_best_accuracy:
    print("Current epoch is the best so far. Saving model...")
    torch.save(mymodel.state_dict(),'current_best_model')
    current_best_accuracy = validate_accuracies[-1]
```

Calculate the average train/validate loss across different batches in this epoch

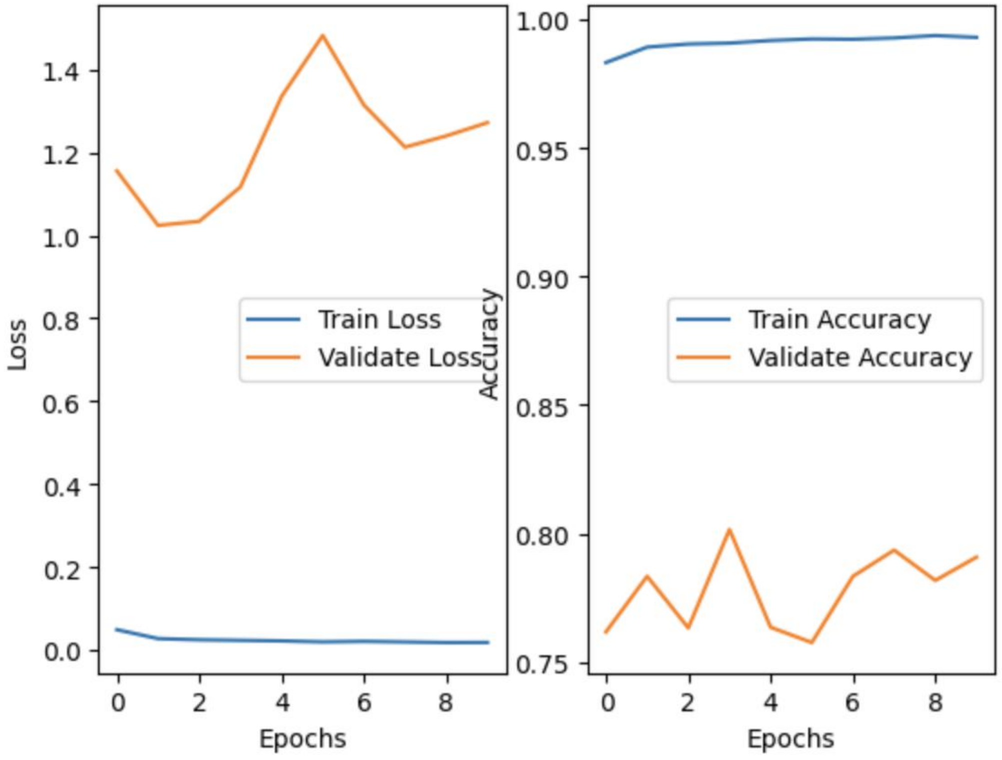
Do the same for the accuracy

Good practice: Print the progress of this epoch, incl. train/validate loss/accuracy

Good practice:
If validate accuracy is the best among all previous epochs, save the model

After Training Loops

Plot the train/validate loss for each epoch




```
fig,axes = plt.subplots(nrows=1,ncols=2)

axes[0].plot(range(N_epochs),losses,label='Train Loss')
axes[0].plot(range(N_epochs),validate_losses,label='Validate Loss')
axes[0].set_xlabel("Epochs")
axes[0].set_ylabel("Loss")

axes[1].plot(range(N_epochs),accuracies,label='Train Accuracy')
axes[1].plot(range(N_epochs),validate_accuracies,label='Validate Accuracy')
axes[1].set_xlabel("Epochs")
axes[1].set_ylabel("Accuracy")
```

After Training Loops

Load the best model encountered in the training loops

```
# create a new model with the same input-output dimension as before
mybestmodel = myMultiLayerPerceptron_TahnActivation(x_train.shape[1],2)

# load the "state_dict" from file into the new model
mybestmodel.load_state_dict(torch.load("current_best_model"))
```

Calculate the test accuracy

```
test_dataloader = DataLoader(test_dataset,batch_size = batch_size,shuffle = True)
test_batch_accuracy = []
for x_batch, y_batch in test_dataloader:
    # pass input data to get the prediction outputs
    prediction_score = mybestmodel(x_batch)

    # Compute metrics (like accuracy) for this batch here
    prediction_label = torch.argmax(prediction_score.detach(),dim=1).numpy()
    test_batch_accuracy.append( np.sum(prediction_label == y_batch.numpy())/x_batch.shape[0])

test_accuracy = np.mean(np.array(validate_batch_accuracy))

print(f"Test accuracy = {np.round(test_accuracy*100,2)}%")
```

Summary of Best Practices

Before the training loop begins

- Define all training hyper-parameters in a single place
- Create (train and validate) DataLoader with shuffle = True

Inside the nested training loop

- Include a validation loop for each epoch
- Record the train/validation loss and metrics
- Print out the progress, incl. epoch, train/validate loss and other metrics
- Save a “best so far” version of the model