Lab 2: PyTorch Basics and Model Training

University of Washington

ECE 596/AMATH 563

Spring 2021

Outline

- Neural Net Training Workflow
- Pytorch Data Type: Tensors
- Graph Computation and Neural Net Models
- Example: Iris Dataset Classification
- Assignment: MNIST Classification

Part 1: Neural Net Training Workflow in Pytorch

Neural Net Workflow Steps

- Prepare Data
- Select Hyperparameter
- Define Model
- Identify Tracked Values
- Train and Validate Model
- Visualization and Evaluation

Neural Net Workflow Steps

- Prepare Data
- Select Hyperparameter
- Define Model
- Identify Tracked Values
- Train and Validate Model
- Visualization and Evaluation

- Data Preparation
  - Define batch size
  - Split train/val/test sets
  - Migrate to Tensors
  - Additional pre-processing (normalization, encoding, etc.)
  - One-hot encoding?

Data-Preprocessing: One-hot Encoding

- One-hot encoding transforms categorical data into one-hot vectors
  - 1 in the index representing your class
  - 0 in all other indices
  - Total dimensions goes from 1 to number of classes
- Use `torch.nn.functional.one_hot()`

Human-Readable

Pet
Cat
Dog
Turtle
Fish
Cat

Machine-Readable

Cat	Dog	Turtle	Fish
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
1	0	0	0

Image source: https://towardsdatascience.com/stop-one-hot-encoding-your-categorical-variables-bbb0fba89809

Neural Net Workflow Steps

- Prepare Data
- Select Hyperparameter
- Define Model
- Identify Tracked Values
- Train and Validate Model
- Visualization and Evaluation

- Hyperparameter Selection
  - Network size and type
  - Learning Rate
  - Regularizers and strength
  - Define loss function and optimizer
  - Other hyperparameters

Neural Net Workflow Steps

- Prepare Data
- Select Hyperparameter
- Define Model
- Identify Tracked Values
- Train and Validate Model
- Visualization and Evaluation

- Model Definition
  - Network Type
  - Network Parameters/Layers
  - Output value(s) and dimensions
  - Forward() Function

Neural Net Workflow Steps

- Prepare Data
- Select Hyperparameter
- Define Model
- Identify Tracked Values
- Train and Validate Model
- Visualization and Evaluation

- Values to Track
  - Training Loss
  - Validation Loss
  - Other relevant values
- Create blank placeholders to populate during training
  - Empty lists, arrays, or tensors

Neural Net Workflow Steps

- Prepare Data
- Select Hyperparameter
- Define Model
- Identify Tracked Values
- Train and Validate Model
- Visualization and Evaluation

- Train Model
  - Calculate loss on training set
  - Backpropagate gradients
  - Update weights
- Validate Model
  - Calculate error on validation or test set
  - Do not update weights
- Save losses in placholders

Neural Net Workflow Steps

- Prepare Data
- Select Hyperparameter
- Define Model
- Identify Tracked Values
- Train and Validate Model
- Visualization and Evaluation

- Visualize Training Progress
  - Plot training and validation losses over course of training
  - Do curves converge?
  - Does loss go up over time?
- Evaluate model
  - Confusion matrix
  - Generate samples
  - Identify model weaknesses

Part 2: Tensors

Torch Tensors

- Main data structure for PyTorch
- Like numpy arrays, but optimized for machine learning
  - Can be migrated to or stored on GPUs
  - Optimized for automatic differentiation
- Three main attributes:
  - Shape – size of each dimension
  - Datatype – form of each entry (float, int, etc.)
  - Device – cpu or cuda (gpu)

Tensor Initialization

- Can create tensor from existing data
  - torch.Tensor([[1, 2], [3,4]])
  - torch.Tensor(np_array)
- Can generate tensor with random or fixed values
  - torch.ones(shape)
  - torch.zeros(shape)
  - torch.rand(shape)

Tensor Operations

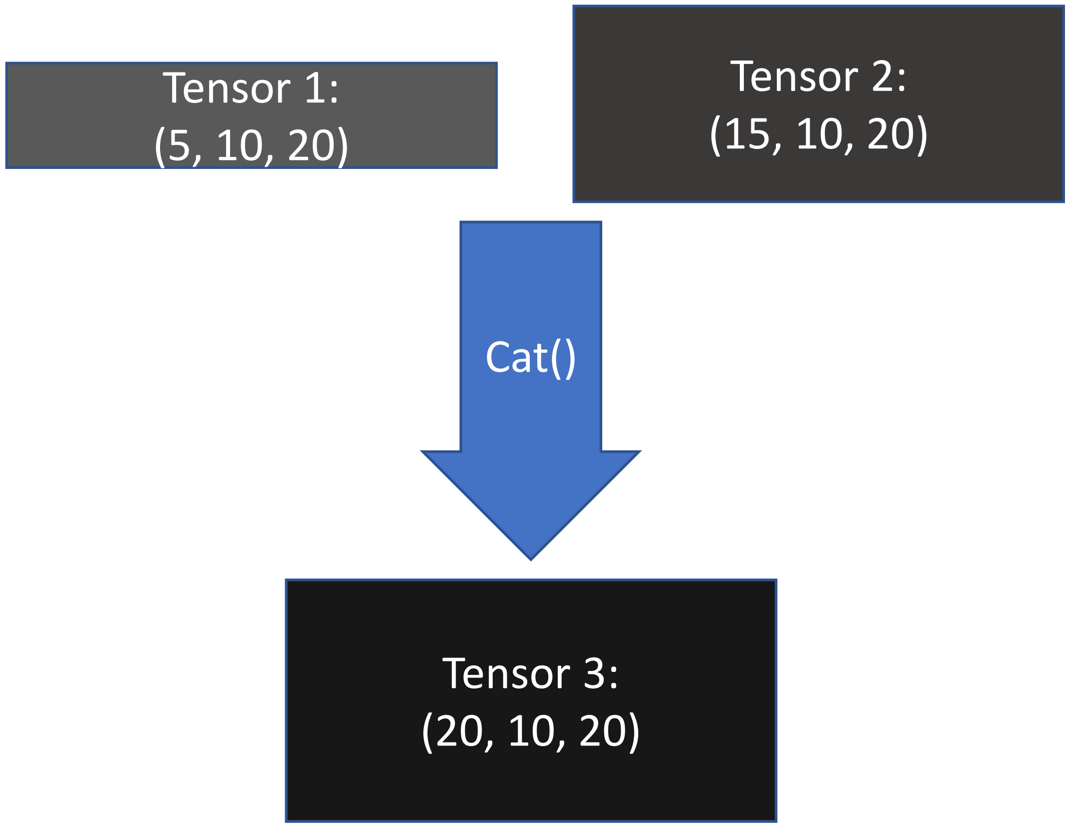
Most numpy operations are identical for tensors

- Indexing and slicing
  - e.g. tensor[:2, 3:5] selects the first 2 rows and 4th and 5th column of tensor
- Elementwise addition, subtraction, multiplication
- Matrix/tensor multiplication: tensor1.matmul(tensor2) OR tensor1@tensor2
  - If tensors have 3 dimension, the first dimension will be treated as the batch (so matmul will be conducted on the other 2 dimensions)

n x d1 x d2 @ n x d2 x d3 = n x d1 x d3

Tensor Operations

- Concatenate tensors using torch.cat()
  - Input should be list or tuple of tensors
  - Can specify dimension over which concatenation occurs (using dim = ?)
  - All other dimensions must be identical



Part 3: Graph Computation and Neural Net Models

Neural Net models in PyTorch

- Base class: nn.Module
- Two primary features of base class:
  - Parameters
  - Forward
- Common PyTorch Layers:
  - Linear
  - Activation Functions (ReLU, tanh, etc.)
  - Dropout
  - RNN
  - Convolution

Neural Network Models

```python
class ExModel(nn.Module):
  '''
  Example model: A simple, two-layer NN with ReLU activation.
  Input dimension is 100, output dimension is 1.
  '''
  def __init__(self):
    super(ExModel, self).__init__()
    self.layer1 = nn.Linear(100, 50)
    self.act1 = nn.ReLU()
    self.output_layer = nn.Linear(50, 1)

  def forward(self, x):
    x = self.layer1(x)
    x = self.act1(x)
    output= self.output_layer(x)
    return output


model = ExModel()
print(model)
```

- Initialization
  - Define layers, parameters of your network
  - The parameters of all layers in the network are included in the parameters of the overall network
- Forward()
  - Defines how the network processes input- called using model(input). Use layers/parameters defined above
  - Never use model.forward(input)

Computational graphs

- PyTorch generates a computational graph every time a parameter or variable with requires_grad is operated on
- The graph is used to back-propagate errors and update the parameters

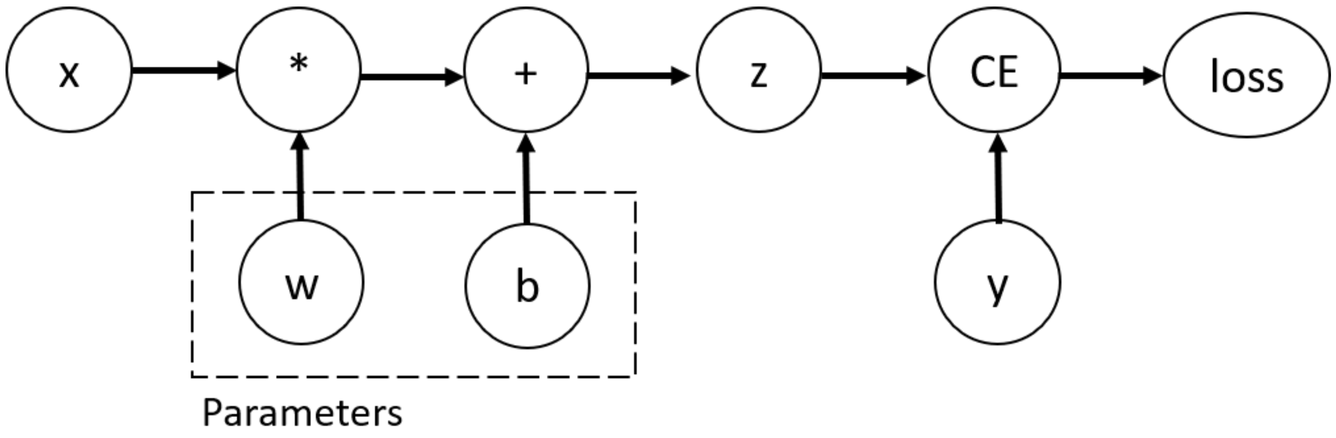


Image source: https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html

Loss functions and Optimizers

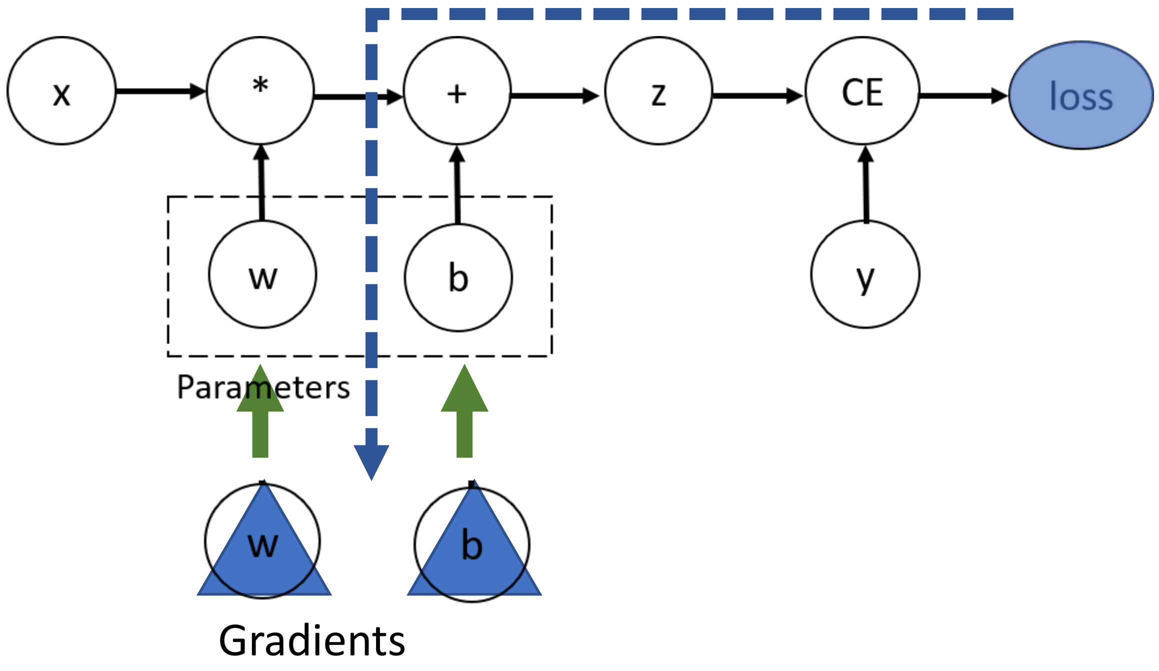
- The loss function defines how you penalize errors
  - nn.MSELoss() for regression
  - nn.NLLLoss() or nn.CrossEntropy() for classification
- Optimizer defines how you update the parameter weights given the gradients given by the loss
  - Stochastic Gradient Descent (optim.SGD)
  - Momentum-based, adaptive approach- Adam (optim.Adam)

- Optimizer needs to be given the learning rate and the model parameters to update

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

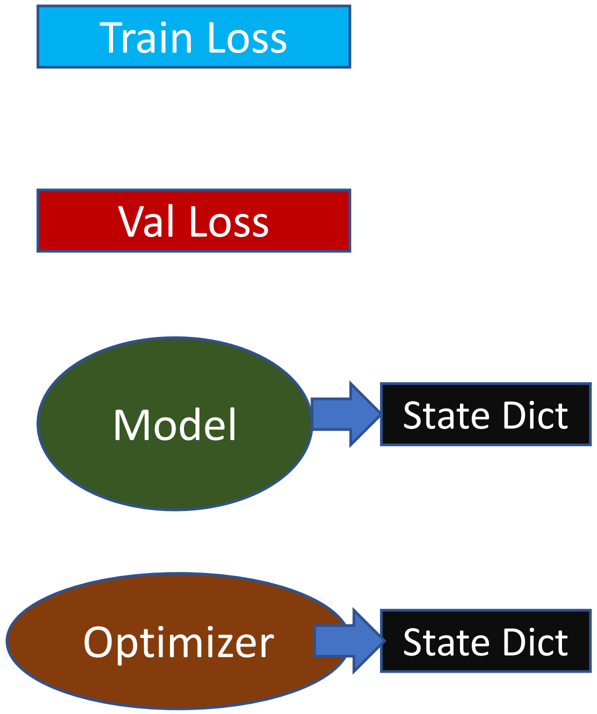
Optimization during training

- Each training step, the optimization occurs in 3 steps
  - optimizer.zero_grad() – Resets the accumulated gradients
  - loss.backward() – Back-propagates the gradients to assign the contribution from each parameter
  - optimizer.step() – Updates the parameters based on the gradients, according to the optimization scheme (optimizer)



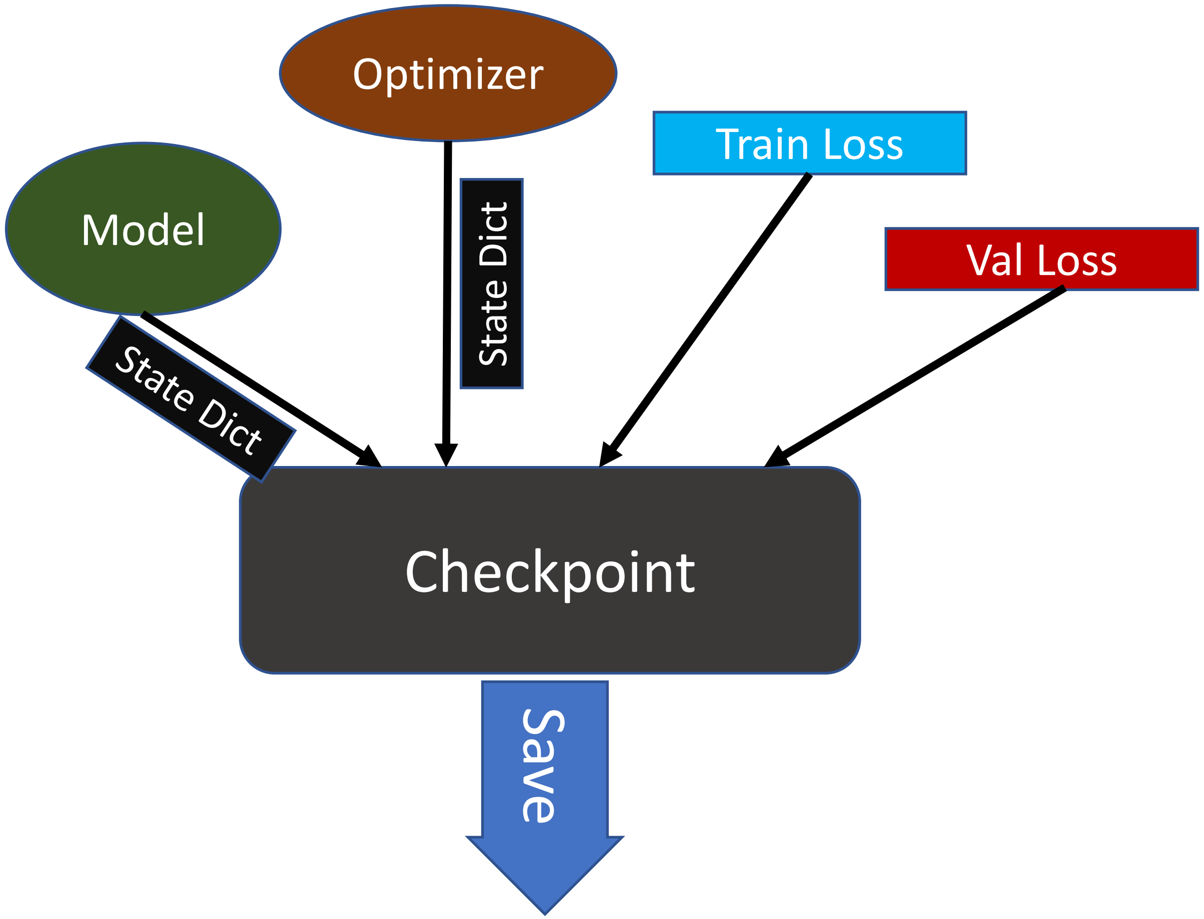
Saving and Loading Models

- Useful quantities to track during training:
    - Training Loss
    - Validation Loss
    - Model state dictionary (parameters)
    - Optimizer state dictionary
- Loading state dictionaries
    - Model: model.load_statedict() loads the saved parameter weights into the model
    - Optimizer: optimizer.load_statedict() loads optimizer state, such as learning rate, momentum, etc.



Saving and Loading Models

- Can use torch.save() and torch.load()
  - Operate similarly to pkl.dump and pkl.load
- Create checkpoints to save all in one file
- Can save every epoch, or define some condition for saving (best loss, every *n* epochs, etc.)




```
ckpt = {'train_losses': train_losses, 'model_weights': model.state_dict(),
        'optimizer_state': optimizer.state_dict()}
```

Example: Iris Dataset Classification

Tutorial adapted from: https://janakiev.com/blog/pytorch-iris/

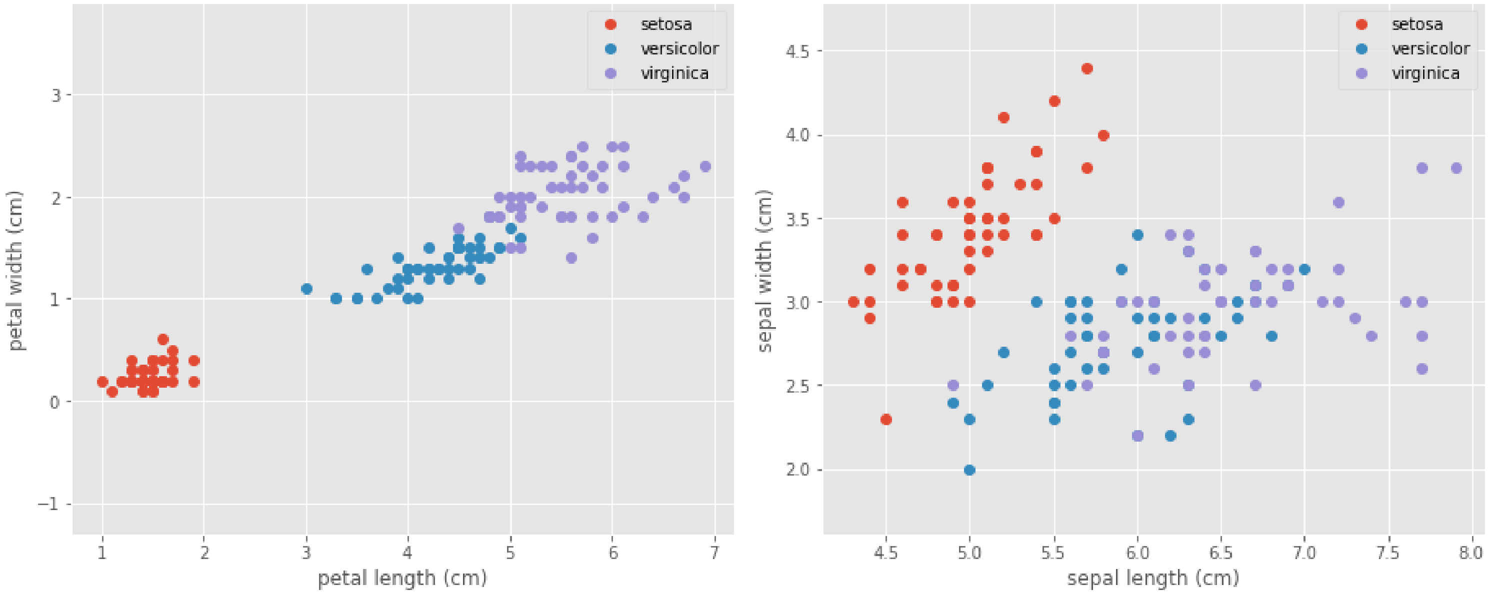
Iris Dataset

- Dataset containing characteristics of 3 different flower types
- Found in sklearn.datasets
- Preprocess data:
  - Train Test split
  - Standard Scaler (Normalize)

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

iris = load_iris()
X = iris['data']
y = iris['target']
names = iris['target_names']
feature_names = iris['feature_names']

# Scale data to have mean 0 and variance 1
# which is importance for convergence of the neural network
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the data set into training and testing
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=2)
```

Iris Dataset

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))
for target, target_name in enumerate(names):
    X_plot = X[y == target]
    ax1.plot(X_plot[:, 0], X_plot[:, 1],
             linestyle='none',
             marker='o',
             label=target_name)
ax1.set_xlabel(feature_names[0])
ax1.set_ylabel(feature_names[1])
ax1.axis('equal')
ax1.legend();

for target, target_name in enumerate(names):
    X_plot = X[y == target]
    ax2.plot(X_plot[:, 2], X_plot[:, 3],
             linestyle='none',
             marker='o',
             label=target_name)
ax2.set_xlabel(feature_names[2])
ax2.set_ylabel(feature_names[3])
ax2.axis('equal')
ax2.legend();
```



- Data Visualization
  - Plotting features against each other
  - Not linearly separable

Model Definition

- 3-layer fully connected neural net
- 1st and 2nd layer have 50 neurons each
- Final layer has 3 neurons
    - Used for classification
- Activation functions can also be called using torch.nn.functional

```python
import torch.nn.functional as F
class Model(nn.Module):
    def __init__(self, input_dim):
        super(Model, self).__init__()
        self.layer1 = nn.Linear(input_dim, 50)
        self.layer2 = nn.Linear(50, 50)
        self.layer3 = nn.Linear(50, 3)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = F.softmax(self.layer3(x), dim=1)
        return x
```

Data, hyperparameters, and Saved Values

- Define loss and optimizer
- Define training epochs and data
- Tracking loss and accuracy at each epoch

```python
import tqdm

EPOCHS  = 100
X_train = torch.from_numpy(X_train).float()
y_train = torch.from_numpy(y_train).long()
X_test  = torch.from_numpy(X_test).float()
y_test  = torch.from_numpy(y_test).long()

loss_list     = np.zeros((EPOCHS,))
accuracy_list = np.zeros((EPOCHS,))
```

```python
model     = Model(X_train.shape[1])
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
loss_fn   = nn.CrossEntropyLoss()
```

Model Training and validation

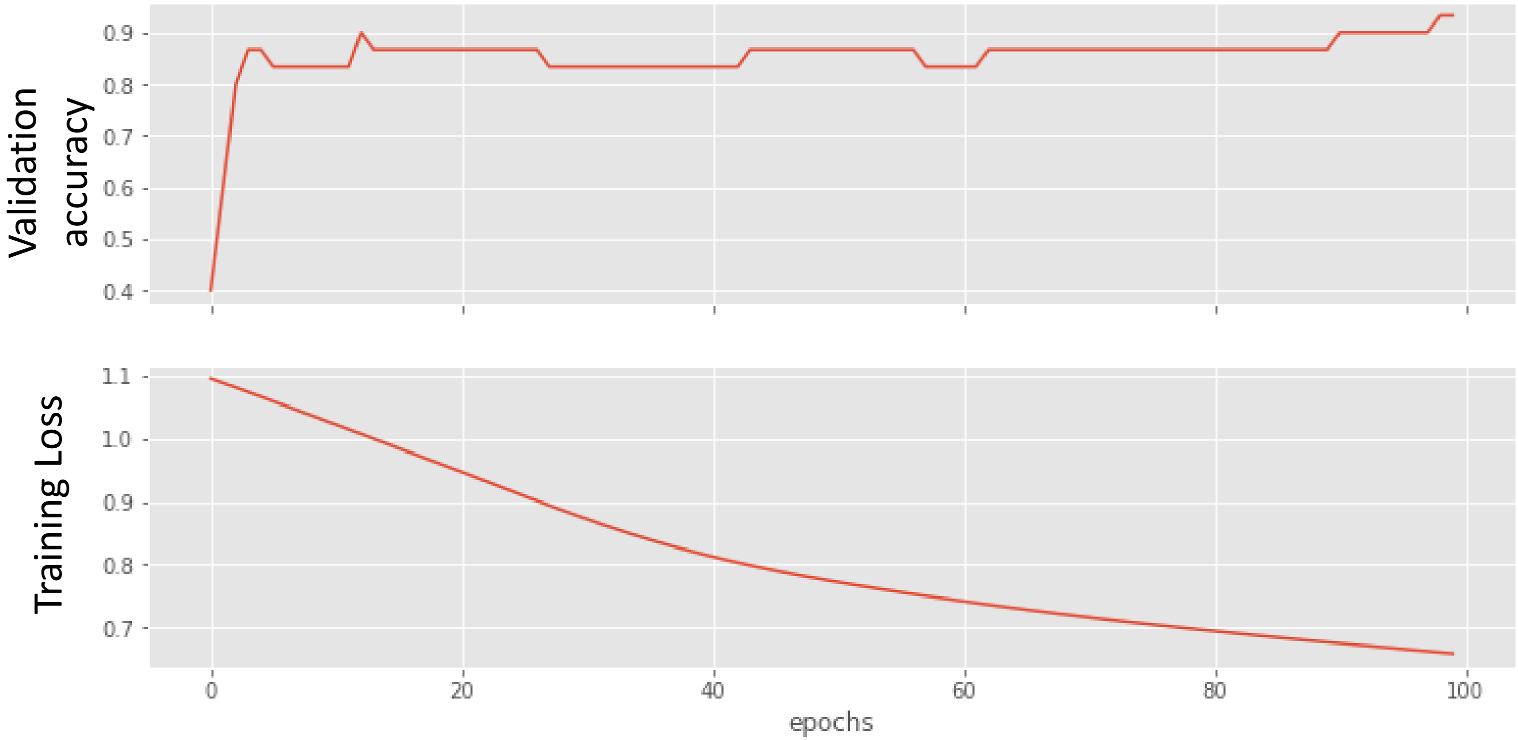
- Output of model gives predictions
- Track training loss as loss
- Check accuracy on validation set

```python
for epoch in tqdm.trange(EPOCHS):
    y_pred = model(X_train)
    loss = loss_fn(y_pred, y_train)
    loss_list[epoch] = loss.item()

    # Zero gradients
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    with torch.no_grad():
        y_pred = model(X_test)
        correct = (torch.argmax(y_pred, dim=1) == y_test).type(torch.FloatTensor)
        accuracy_list[epoch] = correct.mean()
```

Plot Validation accuracy and training loss

```
fig, (ax1, ax2) = plt.subplots(2, figsize=(12, 6), sharex=True)

ax1.plot(accuracy_list)
ax1.set_ylabel("validation accuracy")
ax2.plot(loss_list)
ax2.set_ylabel("training loss")
ax2.set_xlabel("epochs");
```

- Loss should go down
- Accuracy should go up



Lab Assignment: MNIST Classification

The MNIST Dataset

- Handwritten digits 0-9
- Labels are the correct values of the digit
- Data consists of grayscale images of fixed size (28x28) – flattens to 784
- Canonical dataset for machine learning

MNIST Classification Task

- Load the MNIST dataset using torchvision.datasets
- Adjust your hyperparameters to achieve a testing accuracy of 90% or better
- Report your hyperparameters selected and resulting accuracy

```python
import torchvision
train_batch_size = #Define train batch size
test_batch_size  = #Define test batch size (can be larger than train batch size)


# Use the following code to load and normalize the dataset
train_loader = torch.utils.data.DataLoader(
  torchvision.datasets.MNIST('/files/', train=True, download=True,
                             transform=torchvision.transforms.Compose([
                               torchvision.transforms.ToTensor(),
                               torchvision.transforms.Normalize(
                                 (0.1307,), (0.3081,))
                             ])),
  batch_size=batch_size_train, shuffle=True)

test_loader = torch.utils.data.DataLoader(
  torchvision.datasets.MNIST('/files/', train=False, download=True,
                             transform=torchvision.transforms.Compose([
                               torchvision.transforms.ToTensor(),
                               torchvision.transforms.Normalize(
                                 (0.1307,), (0.3081,))
                             ])),
  batch_size=batch_size_test, shuffle=True)
```

Hyperparameters to consider

- First things to try
  - Number of layers
  - Neurons in each layer
  - Training Batch Size
  - Learning Rate
  - Optimizer (SGD vs Adam vs other)
  - Activation function (ReLU vs tanh vs sigmoid)
  - Number of training epochs

- If your training accuracy is very high but test loss is still low:
  - Add Dropout layers (will be covered in more detail next week)
  - Add regularization term to loss
  - Stop training early
  - Make network smaller (fewer layers or neurons)