

# 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

# 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 placeholders

# 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)

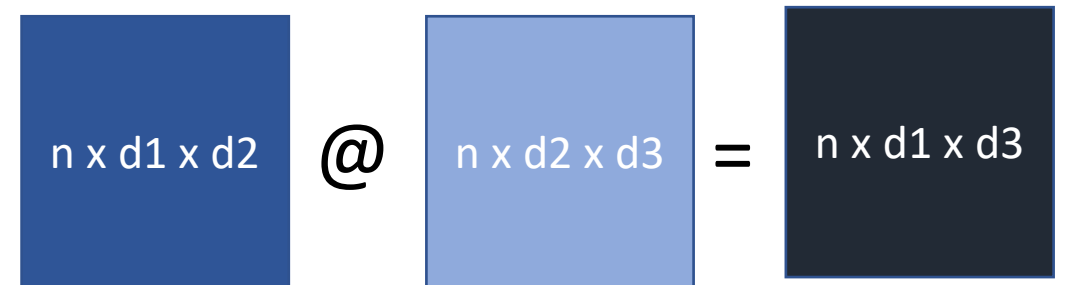
# Tensor Initialization

- 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

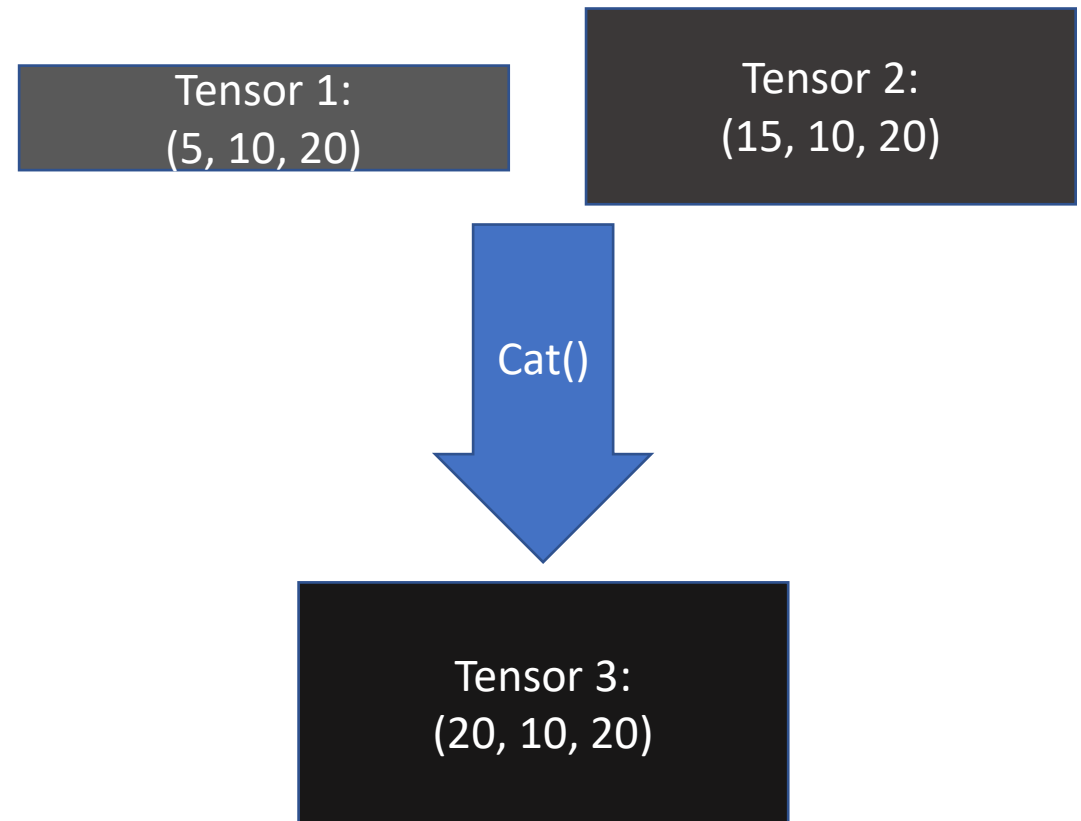
Most numpy operations are identical for tensors

- Indexing and slicing
  - e.g. `tensor[:2, 3:5]` selects the first 2 rows and 4<sup>th</sup> and 5<sup>th</sup> 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)


$$\begin{matrix} n & \times & d1 & \times & d2 \end{matrix} @ \begin{matrix} n & \times & d2 & \times & d3 \end{matrix} = \begin{matrix} n & \times & d1 & \times & d3 \end{matrix}$$

# 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

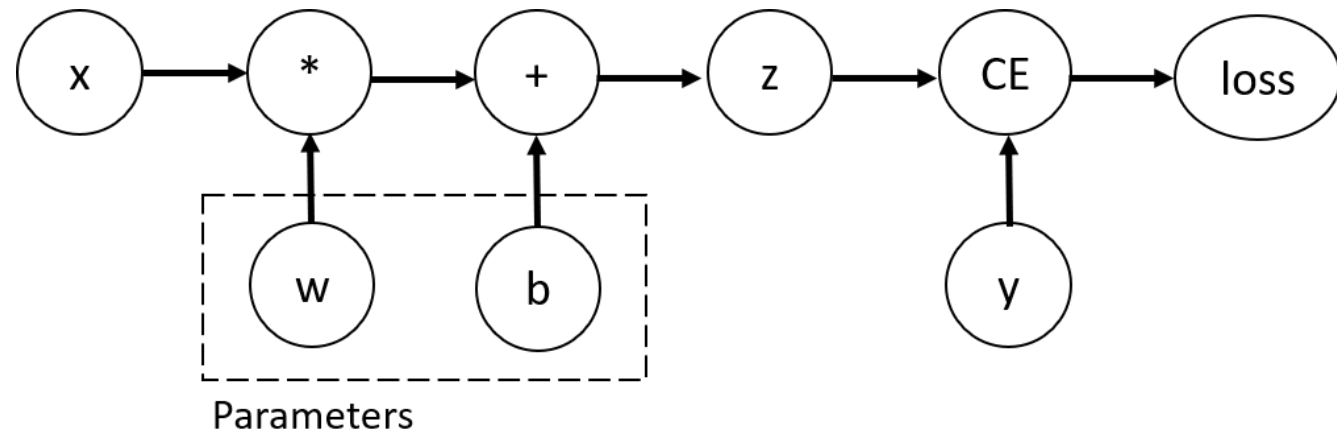
# Neural Network Models

```
1 class ExModel(nn.Module):
2     '''
3     Example model: A simple, two-layer NN with ReLU activation.
4     Input dimension is 100, output dimension is 1.
5     '''
6     def __init__(self):
7         super(ExModel, self).__init__()
8         self.layer1 = nn.Linear(100, 50)
9         self.act1 = nn.ReLU()
10        self.output_layer = nn.Linear(50, 1)
11
12    def forward(self, x):
13        x = self.layer1(x)
14        x = self.act1(x)
15        output = self.output_layer(x)
16        return output
17
18
19 model = ExModel()
20 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



# Loss functions and Optimizers

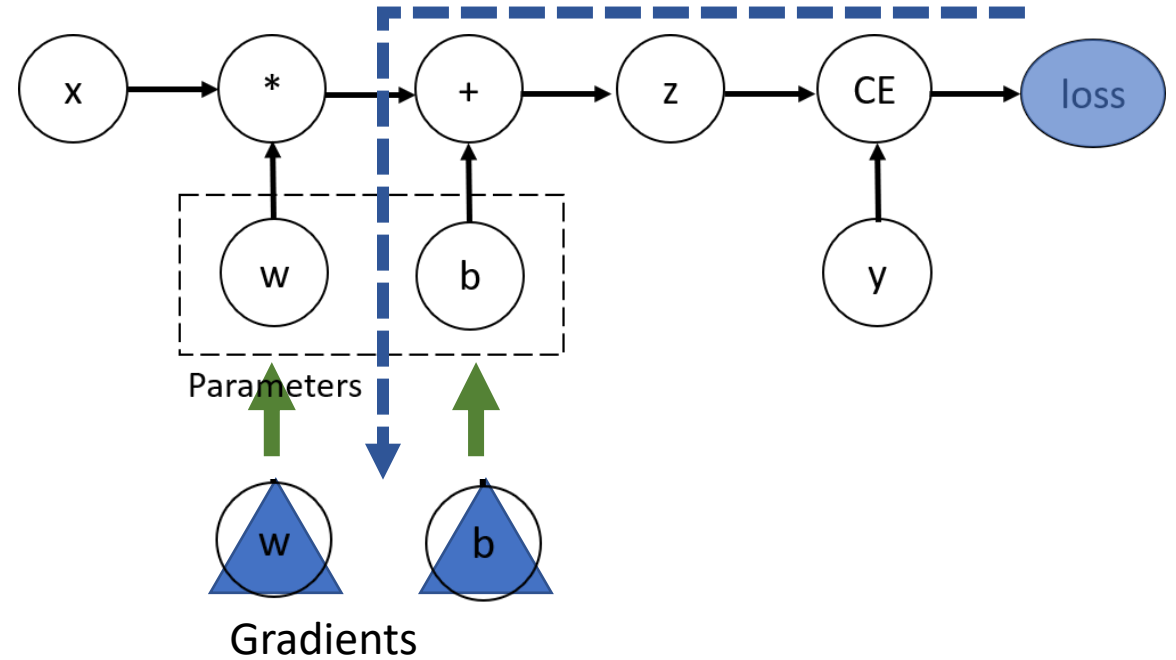
- The loss function defines how you penalize errors
  - `nn.MSELoss()` for regression
  - `nn.NLLLoss()` or `nn.CrossEntropy()` for classification
- Optimizer defines how you update the parameter weights given the gradients given by the loss
  - Stochastic Gradient Descent (`optim.SGD`)
  - Momentum-based,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

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

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

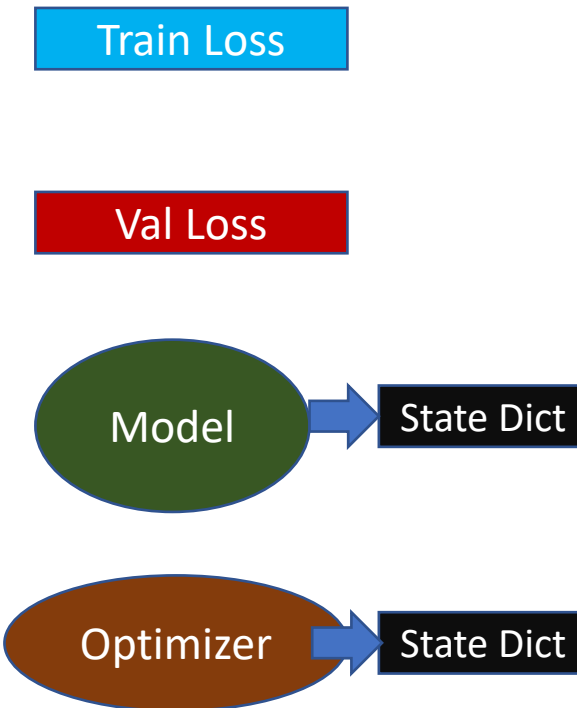
# Optimization during training

- 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)



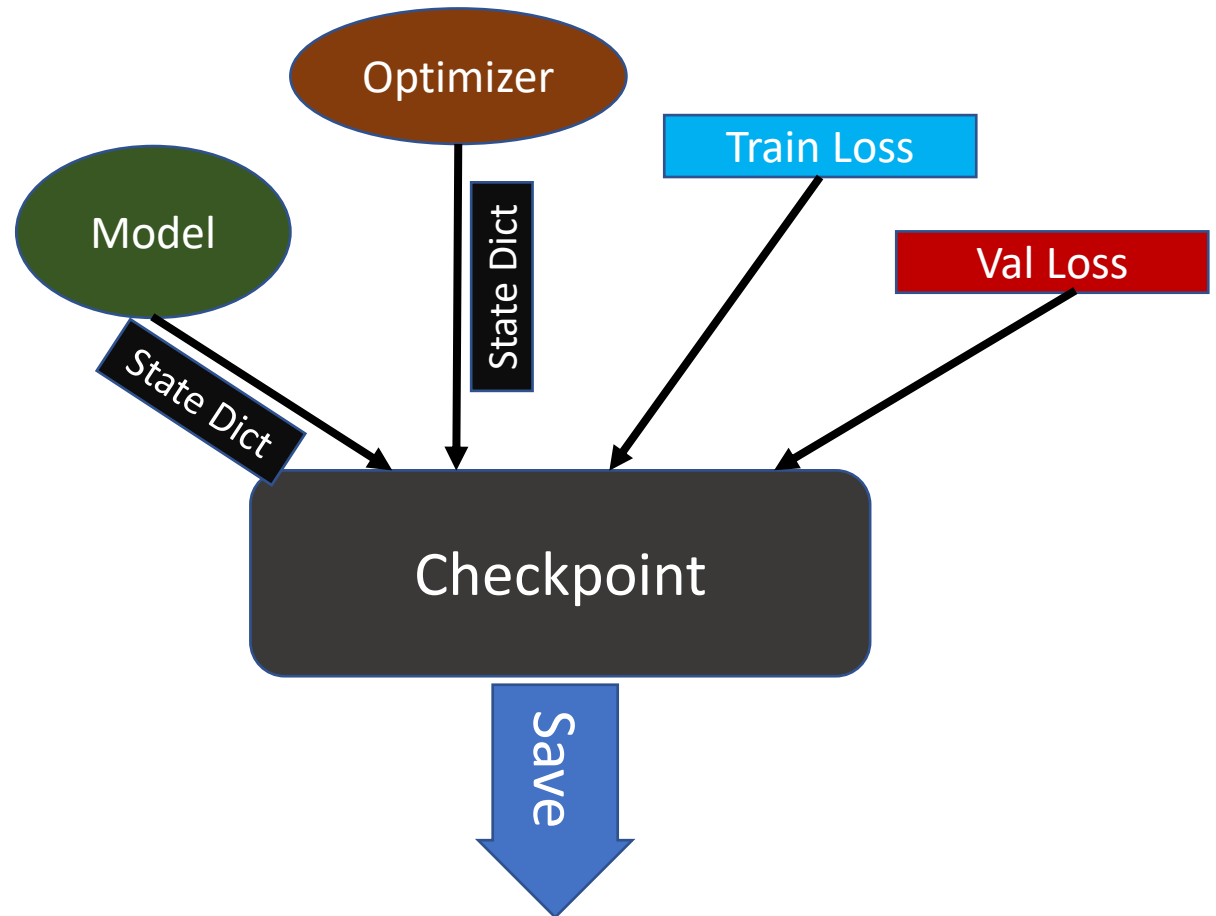
# Saving and Loading Models

- Useful quantities to track during training:
  - Training Loss
  - Validation Loss
  - Model state dictionary (parameters)
  - Optimizer state dictionary
- Loading state dictionaries
  - Model: `model.load_state_dict()` loads the saved parameter weights into the model
  - Optimizer: `optimizer.load_state_dict()` 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

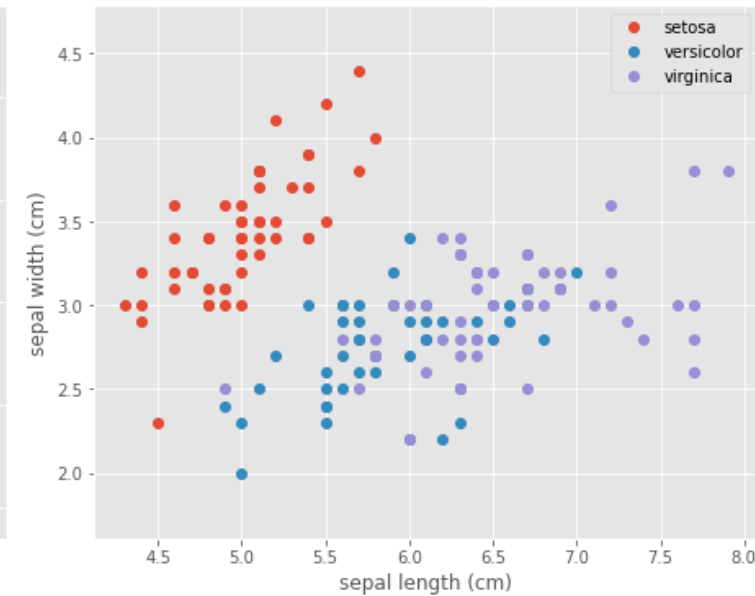
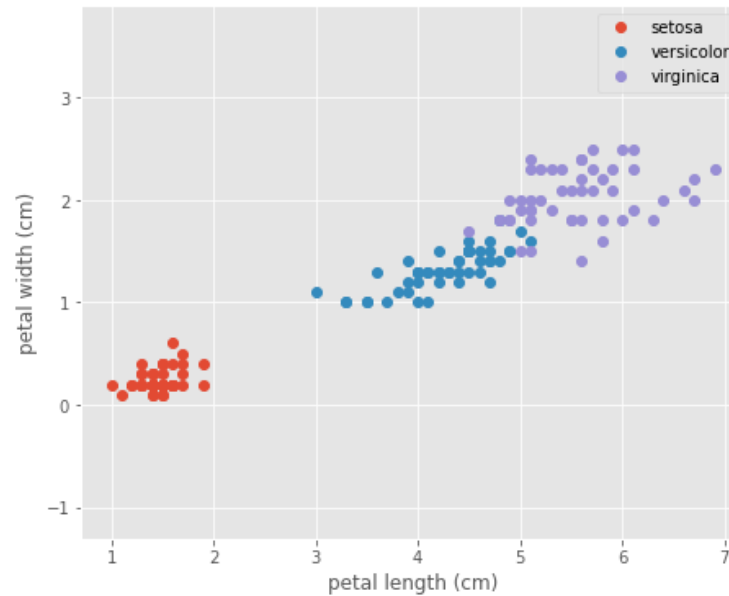
- Dataset containing characteristics of 3 different flower types
- Found in sklearn.datasets
- Preprocess data:
  - Train Test split
  - Standard Scaler (Normalize)



```
1 from sklearn.datasets import load_iris
2 from sklearn.model_selection import train_test_split
3 from sklearn.preprocessing import StandardScaler
4
5 iris = load_iris()
6 X = iris['data']
7 y = iris['target']
8 names = iris['target_names']
9 feature_names = iris['feature_names']
10
11 # Scale data to have mean 0 and variance 1
12 # which is importance for convergence of the neural network
13 scaler = StandardScaler()
14 X_scaled = scaler.fit_transform(X)
15
16 # Split the data set into training and testing
17 X_train, X_test, y_train, y_test = train_test_split(
18     X_scaled, y, test_size=0.2, random_state=2)
```

# Iris Dataset

```
1 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))
2 for target, target_name in enumerate(names):
3     X_plot = X[y == target]
4     ax1.plot(X_plot[:, 0], X_plot[:, 1],
5             linestyle='none',
6             marker='o',
7             label=target_name)
8 ax1.set_xlabel(feature_names[0])
9 ax1.set_ylabel(feature_names[1])
10 ax1.axis('equal')
11 ax1.legend();
12
13 for target, target_name in enumerate(names):
14     X_plot = X[y == target]
15     ax2.plot(X_plot[:, 2], X_plot[:, 3],
16             linestyle='none',
17             marker='o',
18             label=target_name)
19 ax2.set_xlabel(feature_names[2])
20 ax2.set_ylabel(feature_names[3])
21 ax2.axis('equal')
22 ax2.legend();
```



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

# Model Definition

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

```
1 import torch.nn.functional as F
2 class Model(nn.Module):
3     def __init__(self, input_dim):
4         super(Model, self).__init__()
5         self.layer1 = nn.Linear(input_dim, 50)
6         self.layer2 = nn.Linear(50, 50)
7         self.layer3 = nn.Linear(50, 3)
8
9     def forward(self, x):
10         x = F.relu(self.layer1(x))
11         x = F.relu(self.layer2(x))
12         x = F.softmax(self.layer3(x), dim=1)
13         return x
```

# Data, hyperparameters, and Saved Values

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

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

```
1 import tqdm
2
3 EPOCHS     = 100
4 X_train    = torch.from_numpy(X_train).float()
5 y_train    = torch.from_numpy(y_train).long()
6 X_test     = torch.from_numpy(X_test).float()
7 y_test     = torch.from_numpy(y_test).long()
8
9 loss_list   = np.zeros((EPOCHS,))
10 accuracy_list = np.zeros((EPOCHS,))
```

# Model Training and validation

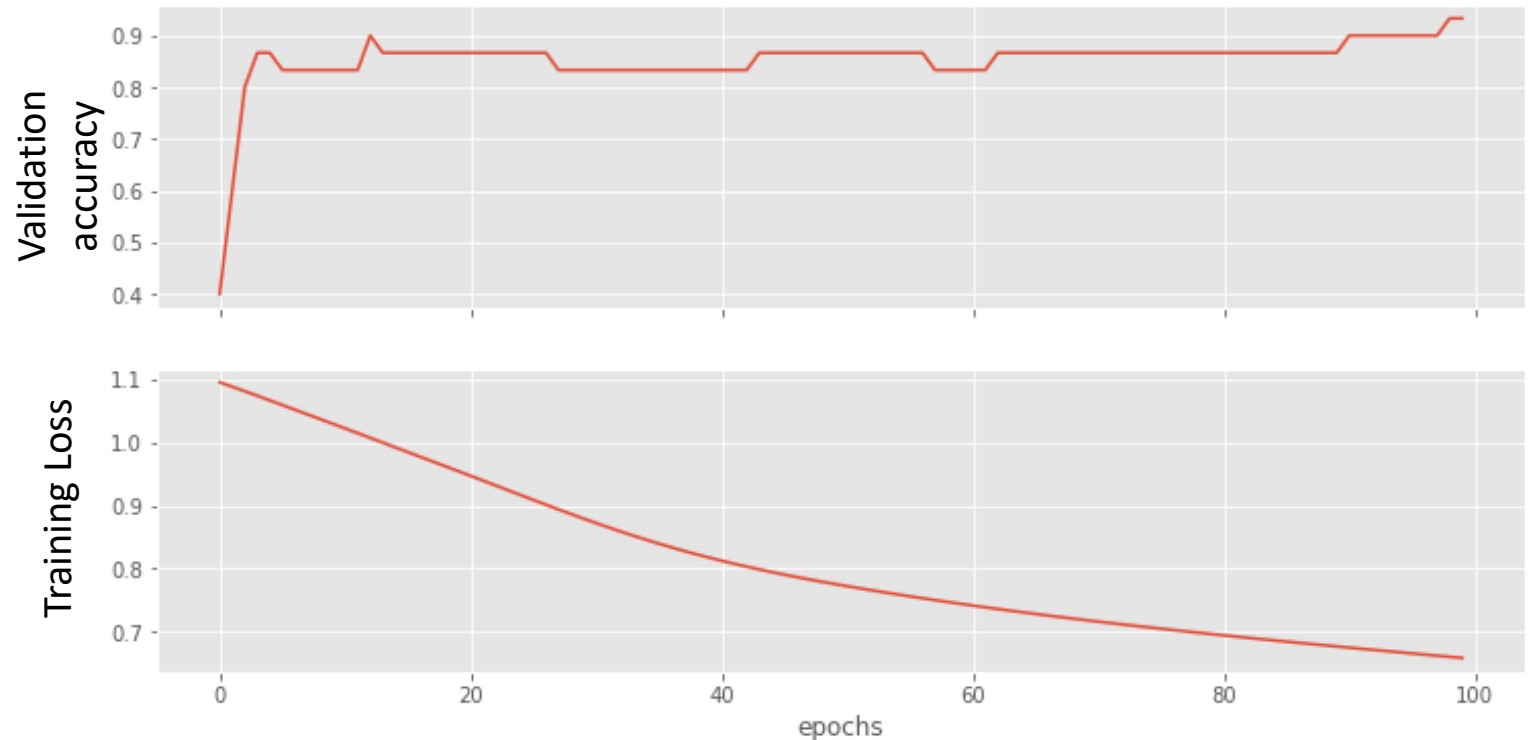
- Output of model gives predictions
- Track training loss as loss
- Check accuracy on validation set

```
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

```
1 fig, (ax1, ax2) = plt.subplots(2, figsize=(12, 6), sharex=True)
2
3 ax1.plot(accuracy_list)
4 ax1.set_ylabel("validation accuracy")
5 ax2.plot(loss_list)
6 ax2.set_ylabel("training loss")
7 ax2.set_xlabel("epochs");
```

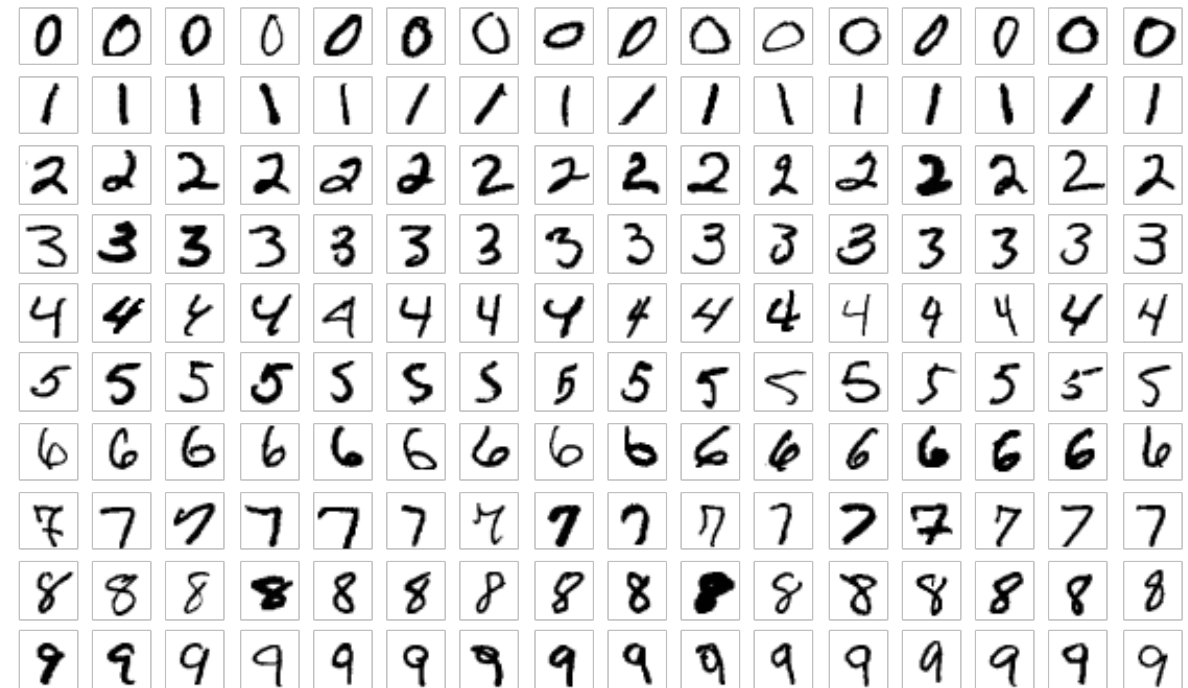
- Loss should go down
- Accuracy should go up



# 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

```
1 import torchvision
2 train_batch_size = #Define train batch size
3 test_batch_size = #Define test batch size (can be larger than train batch size)
4
5
6 # Use the following code to load and normalize the dataset
7 train_loader = torch.utils.data.DataLoader(
8     torchvision.datasets.MNIST('/files/', train=True, download=True,
9                               transform=torchvision.transforms.Compose([
10                                   torchvision.transforms.ToTensor(),
11                                   torchvision.transforms.Normalize(
12                                       (0.1307,), (0.3081,))
13                               ])),
14     batch_size=batch_size_train, shuffle=True)
15
16 test_loader = torch.utils.data.DataLoader(
17     torchvision.datasets.MNIST('/files/', train=False, download=True,
18                               transform=torchvision.transforms.Compose([
19                                   torchvision.transforms.ToTensor(),
20                                   torchvision.transforms.Normalize(
21                                       (0.1307,), (0.3081,))
22                               ])),
23     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)