EE-105
NN training on MNIST

3/26/2025

Objective

- The objective of this lab session is to introduce students to the foundational concepts of building and evaluating neural networks using PyTorch.
- By the end of this lab, students will understand how to:
  - construct a simple artificial neuron,
  - Train on the MNIST dataset
  - Inference on a custom image
  - analyze training dynamics to visually distinguish between underfitting and overfitting models.

SETUP

- For windows users:
  - Double click on the setup_windows file
- For Mac users:
  - Open a terminal from the “mod-4-nn-labs” folder
  - Type chmod +x setup_linux_1.bash (you only need to do this once to give the file permission to run).
  - Run it by typing: bash setup_linux_1

Checkpoint 1

- Import libraries
- Forward pass Through the Neuron
- Understand Perceptrons

Checkpoint 1 - torch

- torch.nn.Module, torch.nn.Parameter

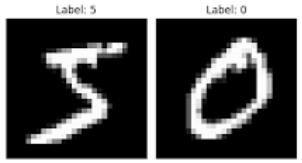
```
#Define the Neuron class
class SimpleNeuron(nn.Module):
    def __init__(self):
        super(SimpleNeuron, self).__init__()
        # Initialize weights and bias
        self.weights = nn.Parameter(torch.randn(3))  # Random initial weights for the inputs
        self.bias = nn.Parameter(torch.randn(1))     # Random initial bias

    def forward(self, x):
        # Compute the weighted sum of inputs plus the bias
        return torch.sum(x * self.weights) + self.bias
```

- activation function: nn.ReLU, nn.Sigmoid, nn.Tanh
- Perceptrons: Using simple NN for a decision making

Checkpoint 2

- Data Loading
- MNIST
- Mini batch gradient descent

Checkpoint 2 - Data



- MNIST: LeCun (1998)
- torchvision.transforms: useful data transformation functions
- torch.utils.data.DataLoader: used everywhere, gives data batches
  - Use different dataloaders for train and test
  - Epochs: number of batches to go through the dataset once

Checkpoint 3

- Defining Neural Networks
- Loss functions and optimizers
- Training and Eval

Checkpoint 3: NN setup

- nn.Linear: both weights and biases included
- nn.CrossEntropyLoss: used for multi class classification
- optim.SGD: Stochastic gradient descent

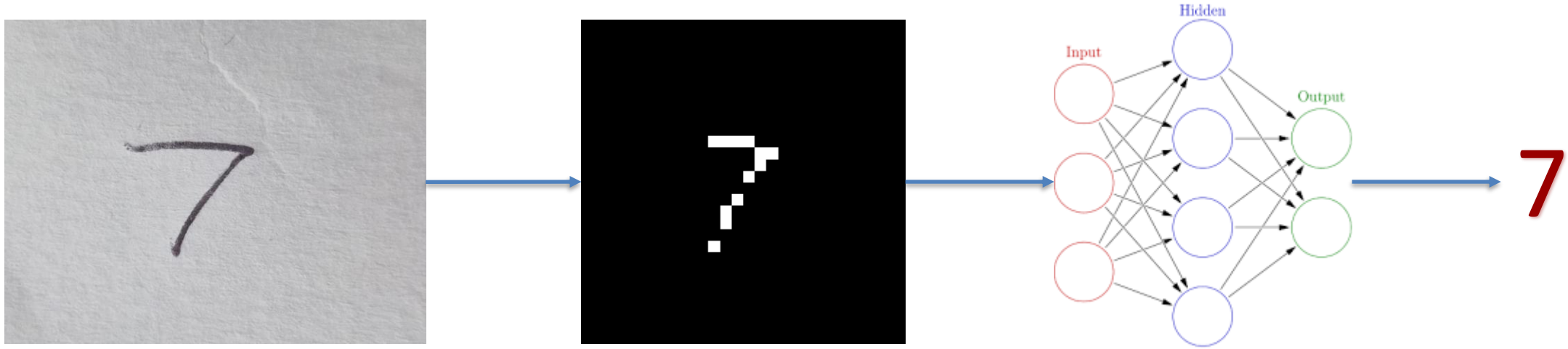
Checkpoint 3: NN training

```
optimizer.zero_grad()  # Clear previous gradients

# Forward pass
outputs = model(images)

# Compute the loss
loss = criterion(outputs, labels)

# Backward pass and optimization
loss.backward()
optimizer.step()
```

Checkpoint 4

- Evaluate simple NN on custom image
- Distribution shifts
- prediction

Checkpoint 4: custom NN inference



Checkpoint 5

- Train 1 hidden layer NN
- Train CNN
- Eval trained CNN
- Experiment with different parameters

Checkpoint 5: Better architectures

```python
#Define the neural network with a hidden layer
class SimpleNetWithHiddenLayer(nn.Module):
    def __init__(self):
        super(SimpleNetWithHiddenLayer, self).__init__()
        self.hidden = nn.Linear(28 * 28, 128)  # Input si
        self.relu = nn.ReLU()                  # ReLU acti
        self.output = nn.Linear(128, 10)       # Output la

    def forward(self, x):
        x = x.view(-1, 28 * 28)  # Flatten the input imag
        x = self.relu(self.hidden(x))  # Apply hidden lay
        x = self.output(x)             # Apply output lay
        return x
```

```python
# Define a simple CNN model
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # First convolutional layer
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, stride=1, padding=1)
        self.relu = nn.ReLU()  # ReLU activation function
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)  # Max pooling layer

        # Second convolutional layer
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)

        # Fully connected layers
        self.fc1 = nn.Linear(32 * 7 * 7, 128)  # Fully connected layer
        self.fc2 = nn.Linear(128, 10)  # Output layer (10 classes for digits 0-9)

    def forward(self, x):
        x = self.relu(self.conv1(x))  # Apply first convolution + ReLU
        x = self.pool(x)  # Apply max pooling
        x = self.relu(self.conv2(x))  # Apply second convolution + ReLU
        x = self.pool(x)  # Apply max pooling again
        x = x.view(-1, 32 * 7 * 7)  # Flatten the output for the fully connected layer
        x = self.relu(self.fc1(x))  # Apply first fully connected layer + ReLU
        x = self.fc2(x)  # Apply output layer
        return x
```

Checkpoint 6

- Under fitting
  - "Simpler" model, fewer epochs
- Overfitting
  - "More complicated" model, more epoch