Homework #5
EE 541: Fall 2023

Name: Nissanth Neelakandan Abirami
USC ID: 2249203582
Instructor: Dr. Franzke

2. In this problem you manually train a multilayer perceptron (MLP) deep neural network using only numpy. Do not use tf.keras, PyTorch, scikitlearn, etc. – use only numpy. The purpose of this problem is to understand backprop and its implementation before we hand the calculation to PyTorch Autograd.
This problem uses the same MNIST image classification problem as the previous homework. So you should build upon the Python code you wrote for the feedforward (inference) processing. The file mnist traindata.hdf5 contains 60,000 images in the key xdata and the corresponding labels in the key ydata. Split into 50,000 training images and 10,000 validation images.
Create a neural network for classification. It must have 784 neurons in the input layer and 10 neurons in the output layer. You may use any number of intermediate (hidden) layers with any number of neurons. It must be a fully connected MLP (don't use convolutional or other layers).
Construct your MLP with:
a. Hidden layer activations: Try tanh and ReLU. Write your own functions for these, and their derivatives. Note: derivative of ReLU at 0 can be any number in [0,1].
b. Cost function: Cross entropy.
c. Optimizer: Stochastic gradient descent. Try 3 different values for learning rate. Momentum is optional. Remember that you can only use numpy, so momentum complicates the implementation.
d. Minibatch size: Your choice. Pick a number between 50 and 1000 that divides by 50,000 (i.e., don't pick 123). Average the gradients for weight and bias updates over each minibatch – e.g., minibatch size of 100 means 50,000/100 = 500 updates per epoch.
e. Regularizer: Your choice. If you choose to use it be mindful when calculating cost gradient.
f. Parameter initialization: Your choice. Don't initialize weights with zeroes. You can use random values like wij ~ U[0, 1] or wij ~ N(0, 1) or use a standard algorithm like He or Xavier normal.
g. Input preprocessing, batch normalization: None
Train the network for 50 epochs using the 50,000 training images. At the end of each epoch complete a forward pass with the validation set and record the accuracy (e.g., 9630 correct out of 10000 means accuracy is 96.3%). DO NOT use the validation set for training. Perform only forward inference with the validation set.
(a) Learning rate decay: Divide the initial learning rate by 2 twice during training. You can do this after any epoch, for example, after epochs 20 and 40. (Do not do this in the middle of an epoch). The final learning rate should be 1/4th of the initial value.
(b) Repeat training and validation for each of the 6 configurations — using 2 different activation functions and 3 initial learning rates. Make 6 plots each with 2 curves – training accuracy

(y-axis) and validation accuracy (y-axis) vs. epoch number (x-axis). Mark the epochs where you divided learning rate.
(c) Choose the configuration (activation and initial learning rate) with the best validation accuracy.
Train this for all 60,000 images = training + validation. Remember to apply a learning rate decay as before. After all 50 epochs are done test with the data in mnist testdata.hdf5 from the previous assignment. Record the final test accuracy.
Remember: Do not touch the test set from mnist testdata.hdf5 until the very end after you decide which configuration is best.
You have a lot of freedom in this problem — welcome to deep learning! This problem does not constrain the number of layers or the number of neurons in each layer. Experiment and don't worry about low accuracy. A reasonable number for final test accuracy is 95%-99%. Submit the following:
• Network configuration – how many layers and how many neurons.
• Batch size
• 3 different values of initial learning rate
• How you initialized parameters
• 6 curves, as described
• Final test accuracy for the best network
Note: Researchers constructed the Fashion-MNIST dataset for classification because MNIST became "too easy" with modern deep learning techniques. Use the in-lecture experiments on Fashion-MNIST to provide guidance for this problem.

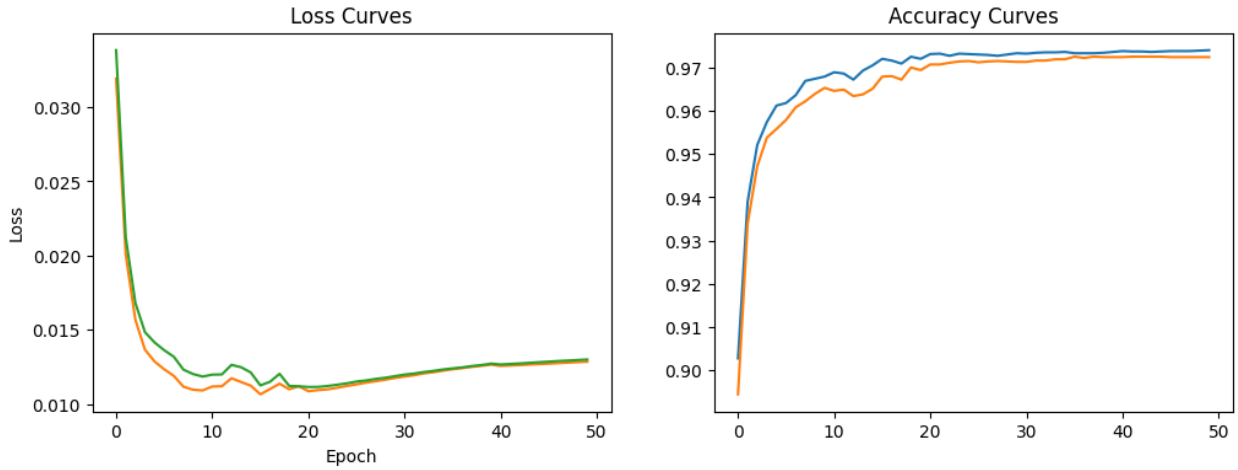
Solutions:

The Tanh function with a learning rate of 0.001 has an accuracy of 97.42 percent,which is the highest among the different learning rate and Activation functions. The model is trained with MNIST dataset and it uses forward inference for only validation sets. This MLP program uses one Input Layer and 3 Hidden layer and 1 Output layer.

Softmax(x): Computes the softmax function for a collection of scores, which is used at the output layer to transform the raw scores of the network into class probabilities.
Relu(x): The activation function for the Rectified Linear Unit (ReLU).
Relu_derivative(x): The ReLU activation function's derivative.
Tanh(x): The activation function of the hyperbolic tangent (tanh).
tanh_derivative(x): The tanh activation function's derivative.

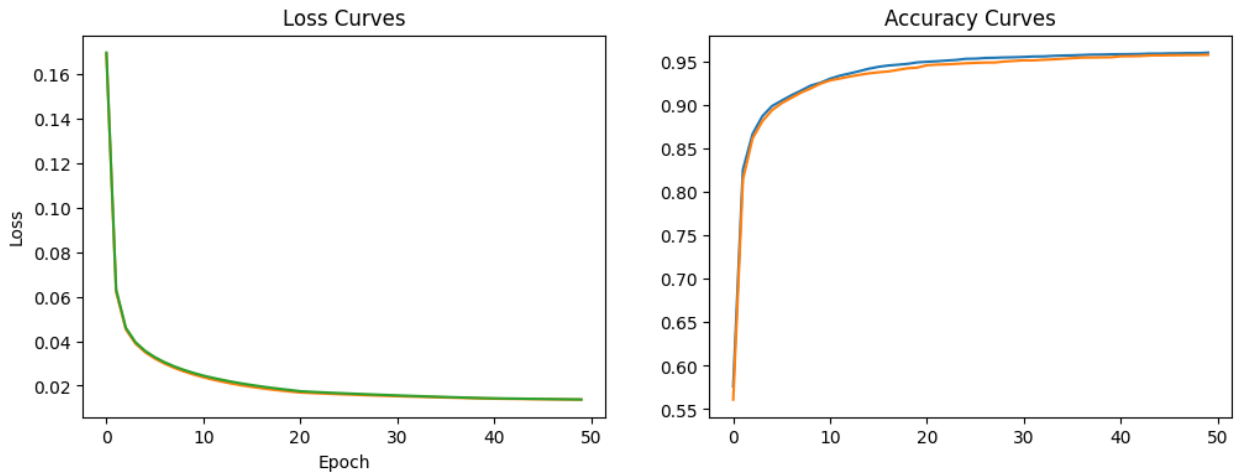
Loss Curves: These graphs depict the training, validation, and test losses with time. They are used to track training progress and detect overfitting.
Accuracy Curves: These curves show the validation and test accuracies with time, demonstrating how well the model performs.

Relu - Learning Rate 0.001



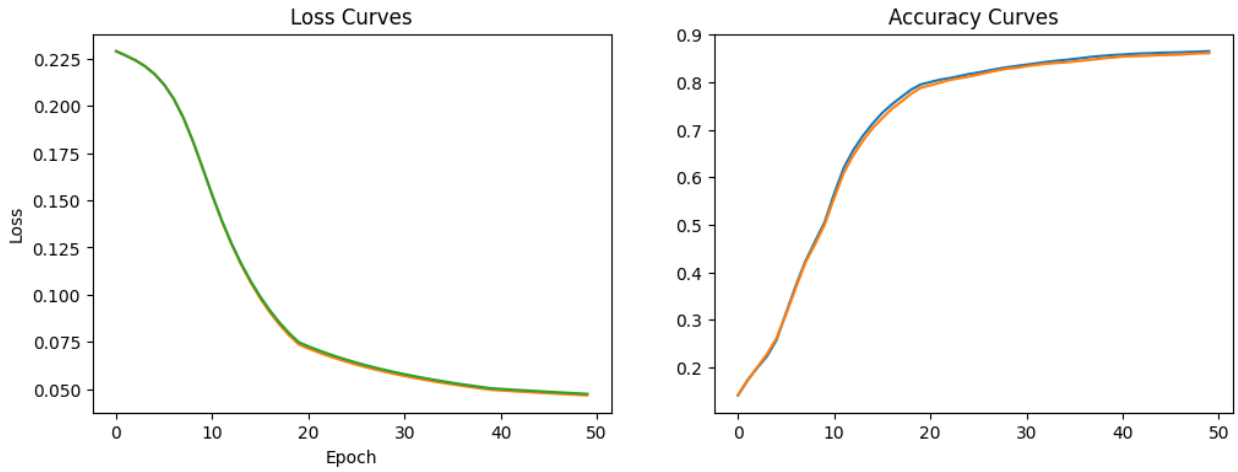

```
Epoch 50/50:
Validation Loss: 0.0141, Test Loss: 0.0137
Validation Accuracy: 0.9720, Test Accuracy: 0.9707
```

Relu - Learning Rate 0.0001



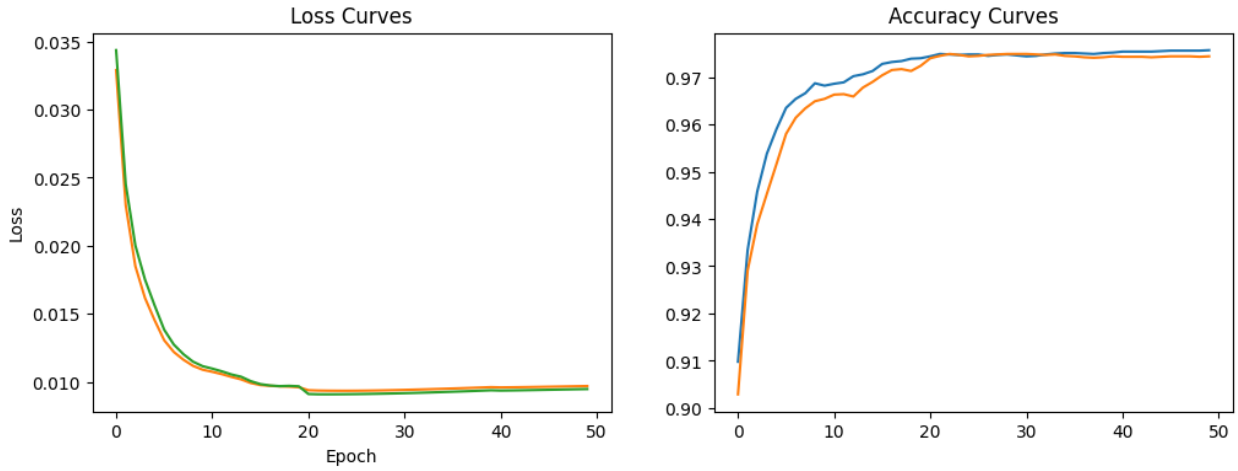

```
Epoch 50/50:
Validation Loss: 0.0144, Test Loss: 0.0147
Validation Accuracy: 0.9604, Test Accuracy: 0.9566
```

Relu - Learning Rate 0.00001



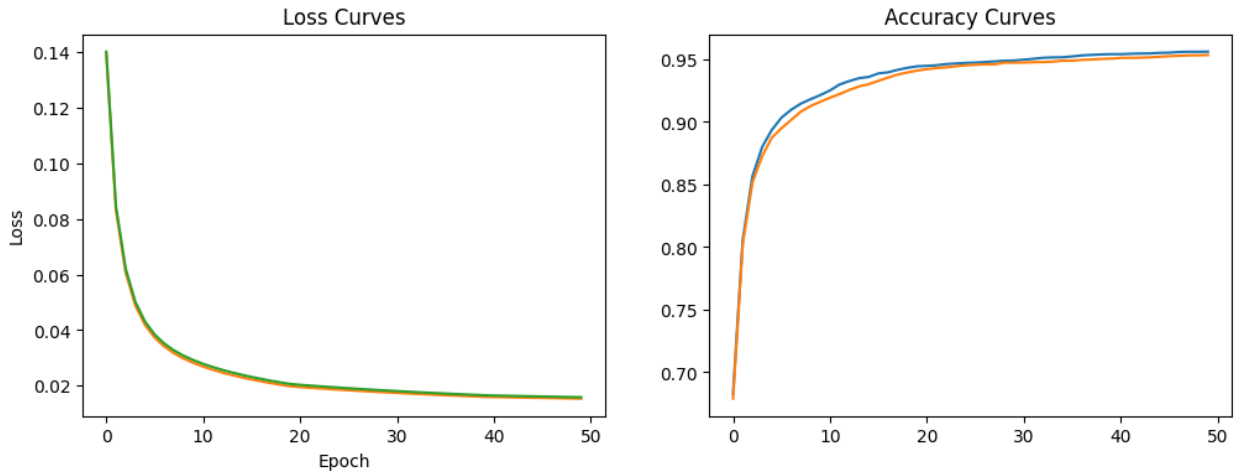

```
Epoch 50/50:
Validation Loss: 0.0485, Test Loss: 0.0499
Validation Accuracy: 0.8590, Test Accuracy: 0.8534
```

Tanh - Learning Rate 0.001



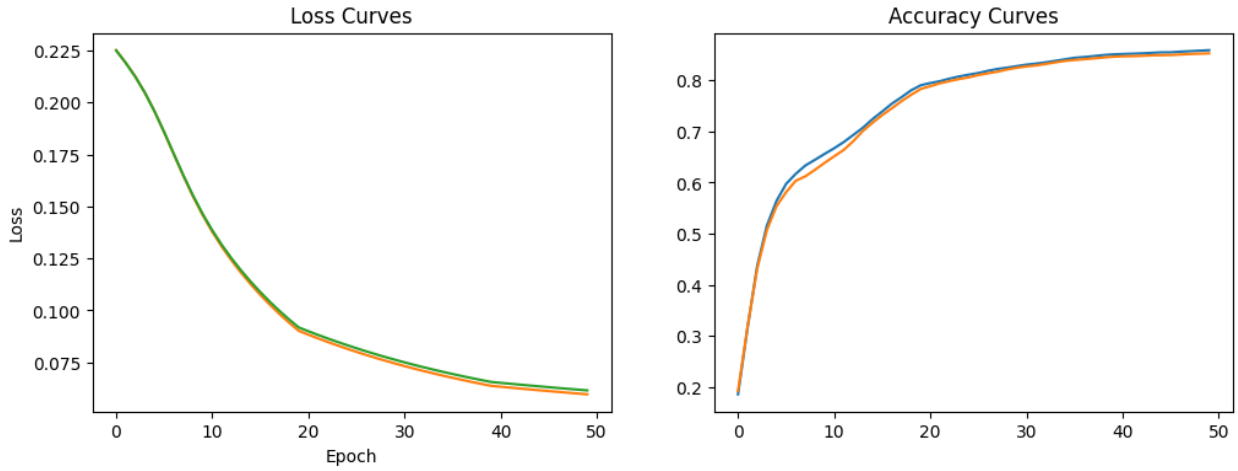

```
Epoch 50/50:
Validation Loss: 0.0099, Test Loss: 0.0094
Validation Accuracy: 0.9755, Test Accuracy: 0.9742
```

Tanh - Learning Rate 0.0001




```
Epoch 50/50:
Validation Loss: 0.0154, Test Loss: 0.0157
Validation Accuracy: 0.9550, Test Accuracy: 0.9532
```

Tanh - Learning Rate 0.00001




```
Epoch 50/50:
Validation Loss: 0.0637, Test Loss: 0.0652
Validation Accuracy: 0.8551, Test Accuracy: 0.8505
```

Batch Size : 500
Learning rates: 0.001, 0.0001, 0.00001

In the provided code, the multi-layer perceptron (MLP) has four hidden layers:
The first hidden layer (h1) has 100 neurons.
The second hidden layer (h2) has 50 neurons.
The third hidden layer (h3) has 25 neurons.
The output layer has 10 neurons, corresponding to the 10 classes in the MNIST dataset.
So, there are a total of 4 hidden layers with varying numbers of neurons in each layer, and the output layer with 10 neurons for classification.

w1, w2, w3, and w4 are weight matrices for each layer. They are initialized with random values drawn from a normal distribution with a mean of 0 and a standard deviation of 0.1.
b1, b2, b3, and b4 are bias vectors for each layer. They are also initialized with random values drawn from a normal distribution with a mean of 0 and a standard deviation of 0.1.

APPENDIX:

Sample code for Tanh:

```
import h5py
import numpy as np
import matplotlib.pyplot as plt

# Load and split the data
data = h5py.File('mnist_traindata.hdf5', 'r')
X = np.asarray(data['xdata'])
Y = np.asarray(data['ydata'])

train_idx = int((5/6) * X.shape[0])
X_train, Y_train = X[:train_idx], Y[:train_idx]
X_valid, Y_valid = X[train_idx:], Y[train_idx:]

test = h5py.File('mnist_testdata.hdf5', 'r')
X_test = np.asarray(test['xdata'])
Y_test = np.asarray(test['ydata'])

# Activation functions and their derivatives
def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=1, keepdims=True)

def relu(x):
    return np.maximum(x, 0)
```

```
def relu_derivative(x):
    return np.where(x > 0, 1, 0)

def tanh(x):
    return np.tanh(x)

def tanh_derivative(x):
    return 1 - x**2

# Calculate accuracy
def accuracy(y_pred, y_true):
    y_pred_max = np.argmax(y_pred, axis=1)
    y_true_max = np.argmax(y_true, axis=1)
    return np.mean(y_pred_max == y_true_max)

# Neural network architecture
input_size, h1, h2, h3, output_size = X_train.shape[1], 100, 50, 25, 10

# Initialize parameters
def init_params(input_size, h1, h2, h3, output_size):
    w1 = np.random.normal(0, 0.1, size=(input_size, h1))
    b1 = np.random.normal(0, 0.1, size=(1, h1))
    w2 = np.random.normal(0, 0.1, size=(h1, h2))
    b2 = np.random.normal(0, 0.1, size=(1, h2))
    w3 = np.random.normal(0, 0.1, size=(h2, h3))
    b3 = np.random.normal(0, 0.1, size=(1, h3))
    w4 = np.random.normal(0, 0.1, size=(h3, output_size))
    b4 = np.random.normal(0, 0.1, size=(1, output_size))
    return w1, b1, w2, b2, w3, b3, w4, b4

w1, b1, w2, b2, w3, b3, w4, b4 = init_params(input_size, h1, h2, h3,
output_size)

# Forward pass
def forward(x, w1, b1, w2, b2, w3, b3, w4, b4):
    a1 = tanh(np.dot(x, w1) + b1)
    a2 = tanh(np.dot(a1, w2) + b2)
    a3 = tanh(np.dot(a2, w3) + b3)
    y_pred = softmax(np.dot(a3, w4) + b4)
    return a1, a2, a3, y_pred
```

```
# Backward pass
def backward(x, y_true, a1, a2, a3, y_pred, w1, w2, w3, w4, b1, b2, b3,
b4, lr=0.001):
    d4 = y_pred - y_true
    d3 = tanh_derivative(a3) * np.dot(d4, w4.T)
    d2 = tanh_derivative(a2) * np.dot(d3, w3.T)
    d1 = tanh_derivative(a1) * np.dot(d2, w2.T)

    grad_w4 = np.dot(a3.T, d4)
    grad_w3 = np.dot(a2.T, d3)
    grad_w2 = np.dot(a1.T, d2)
    grad_w1 = np.dot(x.T, d1)

    grad_b4 = np.sum(d4, axis=0)
    grad_b3 = np.sum(d3, axis=0)
    grad_b2 = np.sum(d2, axis=0)
    grad_b1 = np.sum(d1, axis=0)

    w4 -= lr * grad_w4
    b4 -= lr * grad_b4
    w3 -= lr * grad_w3
    b3 -= lr * grad_b3
    w2 -= lr * grad_w2
    b2 -= lr * grad_b2
    w1 -= lr * grad_w1
    b1 -= lr * grad_b1

    return w1, b1, w2, b2, w3, b3, w4, b4

# Hyperparameters
lr_initial = 0.001
batch_size = 500
n_epochs = 50
lr_decay_epochs = [20, 40]
lr = lr_initial

train_loss, valid_loss, test_loss = [], [], []
train_acc, valid_acc, test_acc = [], [], []
```

```
for epoch in range(n_epochs):
    if epoch in lr_decay_epochs:
        lr /= 2

    if epoch == n_epochs - 1:
        lr = lr_initial / 4

    for j in range(0, len(X_train), batch_size):
        x_batch, y_batch = X_train[j:j+batch_size],
Y_train[j:j+batch_size]
        a1, a2, a3, y_pred = forward(x_batch, w1, b1, w2, b2, w3, b3, w4,
b4)
        w1, b1, w2, b2, w3, b3, w4, b4 = backward(x_batch, y_batch, a1,
a2, a3, y_pred, w1, w2, w3, w4, b1, b2, b3, b4, lr)

    a1, a2, a3, valid_pred = forward(X_valid, w1, b1, w2, b2, w3, b3, w4,
b4)
    valid_loss.append(-np.mean(Y_valid * np.log(valid_pred + 1e-9)))
    valid_accuracy = accuracy(valid_pred, Y_valid)
    valid_acc.append(valid_accuracy)

    a1, a2, a3, test_pred = forward(X_test, w1, b1, w2, b2, w3, b3, w4,
b4)
    test_loss.append(-np.mean(Y_test * np.log(test_pred + 1e-9)))
    test_acc.append(accuracy(test_pred, Y_test))

    print(f"Epoch {epoch+1}/{n_epochs}:")
    print(f"Validation Loss: {valid_loss[-1]:.4f}, Test Loss:
{test_loss[-1]:.4f}")
    print(f"Validation Accuracy: {valid_accuracy:.4f}, Test Accuracy:
{test_acc[-1]:.4f}")


# Plot the learning curves
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(train_loss, label="Training Loss")
plt.plot(valid_loss, label="Validation Loss")
plt.plot(test_loss, label="Test Loss")
plt.xlabel("Epoch")
```

```
plt.ylabel("Loss")
plt.title("Loss Curves")

plt.subplot(1, 2, 2)
plt.plot(valid_acc, label="Validation Accuracy")
plt.plot(test_acc, label="Test Accuracy")
plt.title("Accuracy Curves")

plt.show()
```

Sample code for Relu:

```
import h5py
import numpy as np
import matplotlib.pyplot as plt

# Load and split the data
data = h5py.File('mnist_traindata.hdf5', 'r')
X = np.asarray(data['xdata'])
Y = np.asarray(data['ydata'])

train_idx = int((5/6) * X.shape[0])
X_train, Y_train = X[:train_idx], Y[:train_idx]
X_valid, Y_valid = X[train_idx:], Y[train_idx:]

test = h5py.File('mnist_testdata.hdf5', 'r')
X_test = np.asarray(test['xdata'])
Y_test = np.asarray(test['ydata'])

# Activation functions and their derivatives
def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=1, keepdims=True)

def relu(x):
    return np.maximum(x, 0)

def relu_derivative(x):
    return np.where(x > 0, 1, 0)
```

```
def tanh(x):
    return np.tanh(x)

def tanh_derivative(x):
    return 1 - x**2

# Calculate accuracy
def accuracy(y_pred, y_true):
    y_pred_max = np.argmax(y_pred, axis=1)
    y_true_max = np.argmax(y_true, axis=1)
    return np.mean(y_pred_max == y_true_max)

# Neural network architecture
input_size, h1, h2, h3, output_size = X_train.shape[1], 100, 50, 25, 10

# Initialize parameters
def init_params(input_size, h1, h2, h3, output_size):
    w1 = np.random.normal(0, 0.1, size=(input_size, h1))
    b1 = np.random.normal(0, 0.1, size=(1, h1))
    w2 = np.random.normal(0, 0.1, size=(h1, h2))
    b2 = np.random.normal(0, 0.1, size=(1, h2))
    w3 = np.random.normal(0, 0.1, size=(h2, h3))
    b3 = np.random.normal(0, 0.1, size=(1, h3))
    w4 = np.random.normal(0, 0.1, size=(h3, output_size))
    b4 = np.random.normal(0, 0.1, size=(1, output_size))
    return w1, b1, w2, b2, w3, b3, w4, b4

w1, b1, w2, b2, w3, b3, w4, b4 = init_params(input_size, h1, h2, h3,
output_size)

# Forward pass
def forward(x, w1, b1, w2, b2, w3, b3, w4, b4):
    a1 = relu(np.dot(x, w1) + b1)
    a2 = relu(np.dot(a1, w2) + b2)
    a3 = relu(np.dot(a2, w3) + b3)
    y_pred = softmax(np.dot(a3, w4) + b4)
    return a1, a2, a3, y_pred

# Backward pass
```

```
def backward(x, y_true, a1, a2, a3, y_pred, w1, w2, w3, w4, b1, b2, b3,
b4, lr=0.001):
    d4 = y_pred - y_true
    d3 = relu_derivative(a3) * np.dot(d4, w4.T)
    d2 = relu_derivative(a2) * np.dot(d3, w3.T)
    d1 = relu_derivative(a1) * np.dot(d2, w2.T)

    grad_w4 = np.dot(a3.T, d4)
    grad_w3 = np.dot(a2.T, d3)
    grad_w2 = np.dot(a1.T, d2)
    grad_w1 = np.dot(x.T, d1)

    grad_b4 = np.sum(d4, axis=0)
    grad_b3 = np.sum(d3, axis=0)
    grad_b2 = np.sum(d2, axis=0)
    grad_b1 = np.sum(d1, axis=0)

    w4 -= lr * grad_w4
    b4 -= lr * grad_b4
    w3 -= lr * grad_w3
    b3 -= lr * grad_b3
    w2 -= lr * grad_w2
    b2 -= lr * grad_b2
    w1 -= lr * grad_w1
    b1 -= lr * grad_b1

    return w1, b1, w2, b2, w3, b3, w4, b4

# Hyperparameters
lr_initial = 0.001
batch_size = 500
n_epochs = 50
lr_decay_epochs = [20, 40]
lr = lr_initial

train_loss, valid_lo
ss, test_loss = [], [], []
train_acc, valid_acc, test_acc = [], [], []

for epoch in range(n_epochs):
```

```
    if epoch in lr_decay_epochs:
        lr /= 2

    if epoch == n_epochs - 1:
        lr = lr_initial / 4

    for j in range(0, len(X_train), batch_size):
        x_batch, y_batch = X_train[j:j+batch_size],
Y_train[j:j+batch_size]
        a1, a2, a3, y_pred = forward(x_batch, w1, b1, w2, b2, w3, b3, w4,
b4)
        w1, b1, w2, b2, w3, b3, w4, b4 = backward(x_batch, y_batch, a1,
a2, a3, y_pred, w1, w2, w3, w4, b1, b2, b3, b4, lr)

    a1, a2, a3, valid_pred = forward(X_valid, w1, b1, w2, b2, w3, b3, w4,
b4)
    valid_loss.append(-np.mean(Y_valid * np.log(valid_pred + 1e-9)))
    valid_accuracy = accuracy(valid_pred, Y_valid)
    valid_acc.append(valid_accuracy)

    a1, a2, a3, test_pred = forward(X_test, w1, b1, w2, b2, w3, b3, w4,
b4)
    test_loss.append(-np.mean(Y_test * np.log(test_pred + 1e-9)))
    test_acc.append(accuracy(test_pred, Y_test))

    print(f"Epoch {epoch+1}/{n_epochs}:")
    print(f"Validation Loss: {valid_loss[-1]:.4f}, Test Loss:
{test_loss[-1]:.4f}")
    print(f"Validation Accuracy: {valid_accuracy:.4f}, Test Accuracy:
{test_acc[-1]:.4f}")


# Plot the learning curves
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(train_loss, label="Training Loss")
plt.plot(valid_loss, label="Validation Loss")
plt.plot(test_loss, label="Test Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
```

```
plt.title("Loss Curves")

plt.subplot(1, 2, 2)
plt.plot(valid_acc, label="Validation Accuracy")
plt.plot(test_acc, label="Test Accuracy")
plt.title("Accuracy Curves")

plt.show()
```

1. Backprop Initialization for Multiclass Classification
The softmax function h(·) takes an M-dimensional input vector s and outputs an M-dimensional output vector a as
a = h(s) =1MX m=1 esm es1 es2 … esM
and the multiclass cross-entropy cost is given by
C = − Xn i=1 yi ln ai
where y is a vector of ground truth labels. Define the error (vector) of the output layer as:
δ = ∇sC = ˙A∇aC
where ˙A is the matrix of derivatives of softmax, given as
˙A = dh(s)/ds
(denominator convention with the left-handed chain rule.). Show that δ = a − y if y is one-hot.

EE 541 - A Computational Introduction to Deep Learning

1) G.T

Softmax function

$$a = h(s) = \frac{1}{\sum_{n=1}^{M} e^{s_m}} \begin{bmatrix} e^{s_1} \\ e^{s_2} \\ \vdots \\ e^{s_M} \end{bmatrix}$$

Cross Entropy

$$C = -\sum_{i=1}^{n} y_i \ln a_i$$

Error

$$f = \nabla_S C \approx \dot{A} \nabla_a C$$

$\dot{A}$ (derivative matrix) $= \dot{A}$

$$= \frac{dh(s)}{dt}$$

$$= \begin{bmatrix} \frac{\partial p_1}{\partial s_1} & \cdots & \frac{\partial p_M}{\partial s_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial p_1}{\partial s_M} & \cdots & \frac{\partial p_M}{\partial s_M} \end{bmatrix}$$

$$f = \frac{\partial g}{\partial s_n} = \nabla_s C$$

∂a_n

$$= \frac{\partial C}{\partial a_n} \cdot \frac{\qquad}{\partial s_n}$$

Piazza specified $n \rightarrow$ no of output dim

$M \rightarrow$ no of output class

So $n = M$

(i) $n = M$

From software - it uses M dimensional input vectors and produce M dimensional output vectors.

$$a = \frac{e^{s_n}}{\sum_{} s_i}$$

$$\sum_{n=1} e^{-M}$$

We need to calculate

$$\frac{\partial c}{\partial s_n} = \frac{\partial c}{\partial a_n} \cdot \frac{\partial a_n}{\partial s_n}$$

$$= \left(-\sum_{i=1}^{n} y_i \ln a_i \right) \times \frac{\partial}{\partial a}$$

$$= -y_i \left(\sum_{i=1}^{n} \frac{\partial}{\partial a} \ln a_i \right)$$

$i = 1$ to n

$$= -y_n \times \frac{1}{a_n}$$

$$= -\frac{y_n}{a_n}$$

$$\frac{\partial a_n}{\partial s_n} = \frac{\partial}{\partial s_n}\left[\frac{e^{s_n}}{\sum_{M=1}^{M} e^{s_M}}\right]$$

$$\frac{u}{v} = \frac{vu' - uv'}{v^2}$$

$$= \frac{\sum_{m=1}^{M} e^{s_m} \frac{\partial}{\partial s} e^{s_n} - e^{s_n} \frac{\partial}{\partial s_n} \sum_{m=1}^{M} e^{s_M}}{\left[\sum_{m=1}^{M} e^{s_m}\right]^2}$$

$$= \frac{e^{s_n} \sum_{m=1}^{M} e^{s_M} - e^{s_n} \times e^{s_M}}{}$$

$$\left(\sum_{m=1}^{M} e^{s_m} \right)^2$$

WKT

$$a_n = \frac{e^{s_n}}{\sum_{m=1}^{M} e^{s_m}}$$

$$\frac{e^{s_n}}{\sum_{m=1}^{M} e^{s_m}} \left[\frac{\sum_{m=1}^{M} e^{s_m}}{\sum_{m=1}^{M} e^{s_m}} \right] = \frac{e^{s_n}}{\sum_{m=1}^{M} e^{s_m}}$$

$$= \frac{e^{s_n}}{\sum_{m=1}^{M} e^{s_m}} \left[1 - \frac{e^{s_M}}{\sum_{m=1}^{M} e^{s_m}} \right]$$

$$= a_n \left[1 - \frac{e^{s_M}}{\sum_{m=1}^{M} e^{s_m}} \right]$$

$$\left[\quad \sum_{m=1} \quad \right]$$

$\frac{dL}{dz_n}$ is calculated only when the input and output class index matches if it is not equal then

$$\frac{dL}{dz_n} = 0$$

$$f = \frac{-y_n}{a_n} \times f_n \times \left[1 - \frac{e^{S_m}}{\sum_{m=1}^{M} e^{S_m}} \right]$$

a_m

$$= -y_n \left[1 - \frac{e^{S_m}}{\sum_{m=1}^{M} e^{S_m}} \right]$$

$$= -y_n [1 - a_m]$$

$= -y_n \left[1 - a_n\right]$

(ii) For $n \neq M$

$$\frac{\partial}{\partial s_n} = \sum_{m \neq n} \frac{\partial L}{\partial a_m} \cdot \frac{\partial a_m}{\partial s_n} + \frac{\partial L}{\partial a_n} \cdot \frac{\partial a_n}{\partial s_n}$$

$$= -\sum_{i=1}^{?} \frac{\partial}{\partial a_m}\left[y_i \log a_i\right] \qquad = -\sum_{i=1}^{?} \frac{\partial}{\partial a_n}\left[y_i \log a_i\right]$$

$$= \frac{-y_m}{a_m} \qquad = \frac{-y_n}{a_n}$$

$$= \frac{\partial}{\partial s_n}\left[\frac{e^{s_m}}{\sum_{m=1}^{M} e^{s_m}}\right]$$

$$= \sum_{m=1}^{M} e^{s_m} \cdot \frac{\partial}{\partial s_n}\left(\frac{1}{e^{s_m}}\right) \quad \text{value}$$

$$e^{s_m} \cdot \frac{\partial}{\partial s_n}\left(\sum_{m=1}^{M} e^{s_m}\right) \quad \text{Summation}$$

$$\left(\sum_{m=1}^{M} e^{s_m} \right)^2$$

$$= \frac{0 - e^{s_m} \cdot e^{s_n}}{\left(\sum_{m=1}^{M} e^{s_m} \right)^2}$$

$$= -a_m \cdot a_n$$

$$\frac{dc}{ds_n} \xrightarrow{①} = \frac{dc}{da_m} \cdot \frac{da_m}{ds_n}$$

$$= \frac{-y_m}{a_m} \times -a_m \cdot a_n$$

$$= y_m \cdot a_n$$

$\frac{\partial a_n}{\partial s_n}$ for $m=n$ case

$$= a_n(1 - a_m)$$

$$\frac{\partial L}{\partial a_n} \cdot \frac{\partial a_n}{\partial s_n} = \frac{-y_n}{a_n} \times a_n(1-a_m) \quad \text{②}$$

$$= -y_n(1-a_m)$$

$$\int = \sum_{m \neq n} y_m \cdot a_n - y_n(1-a_m)$$

$$= \sum_{m \neq n} y_m \cdot a_n - y_n + y_n a_m$$

$$= \left(\sum_{m \neq n} y_m \cdot a_n + y_n a_m - y_n \right)$$

b.T y is one hot vector so

Let $\sum_m y = 1$, $\sum_n y = 1$

$\Rightarrow a_n \cdot a_m = y_n$

$\Rightarrow$ assume $a_n = a_m$

$\Rightarrow a_n - y_n$

$$f = a - y$$