EECS-553 Homework 4

Vipransh Sinha

April 8, 2025

Exercise 1

This exercise will focus on training a neural network classifier for the MNIST dataset.
Background: MNIST is a standard digit classification dataset. Given a 28×28 image containing a digit from 0 to 9, our goal is deducing which digit the image corresponds to. The dataset has 60,000 training and 10,000 test examples. The data can be downloaded via

```
from torchvision import datasets, transforms
mnsit = datasets.MNIST(root='./data', train=True, download=True,
transform=transforms.ToTensor())
```

Formatting the data: Let us format the data to obtain a dataset $S = (x_i, y_i)_{i=1}^{N=60,000}$

- **Input:** Each image x is a 28×28 matrix. Apply the following operations to obtain $d = 785$ dimensional input features.
  - Flatten inputs x to vectors of size $28^2 = 784$
  - Standardize input pixel using z-normalization: Scale each pixel i ($1 \leq i \leq 784$) to have zero-mean and unit variance by calculating mean and variance from training samples. In Python terms, this corresponds to the operation $x \rightarrow \frac{x_i - np.mean(x_i)}{np.std(x_i)}$ where x_i represents the i'th pixel. Apply the same normalization during test using mean-variance computed from training data.
  - Add bias variable by concatenating 1 to your input i.e the input become $x \rightarrow \begin{bmatrix} x \\ 1 \end{bmatrix}$. The input dimension becomes $d = 785$.
- **Classification tasks:** MNIST is a multiclass classification problem where each label y is a digit from 0 to 9. We will consider two tasks.
  - **Task A: Binary classification.** We wish to predict whether the digit is greater than 4 or not. The output y is converted to 0 if $0 \leq y \leq 4$ and 1 is $5 \leq y \leq 9$. As loss function, we will employ either squared loss or logistic regression.
  - **Task B: Original multiclass problem.** We wish to predict the digit y from 0 to 9. We will use cross-entropy minimization.

- **Model: Shallow Neural Network.** We will use a shallow neural network with one-hidden layer. The model has the following form:

$$f(x; V, W) = V\text{ReLU}(Wx)$$

  Here $W \in \mathbb{R}^{h\times d}$ is the input layer with h hidden neurons whereas $V \in \mathbb{R}^{K\times h}$ is the output layer. $K = 1$ for Task A and $K = 10$ for Task B. We will use the ReLU activation.

  Initialization: It is critical to initialize neural networks properly. In this assignment, initialize your weight matrices V, W with independent and identical Gaussian distributed entries using He initialization. This means that, set the initial weight matrices $W_o \sim^{i.i.d} \mathcal{N}(0, \frac{2}{d})$ and $V_0 \sim^{i.i.d} \mathcal{N}(0, \frac{2}{h})$.

- **Optimizer:** You are expected to use stochastic gradient descent with **batch size 16**. To keep things simple, you can use constant learning rate. You should tune the learning rate to make sure optimization succeeds. Optimize for 8 epochs over the data i.e use $(N/16) \times 8 = 30,000$ mini-batch iterations. You can do this by monitoring training loss/accuracy as we do not monitor a holdout validation data. Do not worry about choosing the perfect learning rate.

Assignment

Your tasks are as follows. All algorithms should be your own code. No Tensorflow/Pytorch except downloading data.

1. Apply the aforementioned normalization on the training and test data.
2. Suppose the neural network was initialized with $V_0 = W_0 = 0$ rather than randomly. Prove that optimization would get stuck in (V_0, W_0)
3. **Task A:** Binary Classification
   (a) Write a function to compute the gradient $\nabla f(V, W)$ through matrix operations given a minibatch of data $(x_i, y_i)_{i=1}^{B}$.
   (b) Train a neural network classifier with quadratic loss $l(y, f(x)) = \frac{1}{2}(y - f(x))^2$. Plot the progress of the test and training accuracy (y-axis) as a function of the iteration counter t (x-axis). In your figures, you should report the performance at every 100 iterations or higher frequency. Report the final test accuracy for the following choices
      - h=5
      - h=40
      - h=200

   Comment on the role of hidden units h on accuracy and the ease of optimization.

(c) Train a neural network classifier with logistic loss, namely $l(y, f(x)) = -y \log(\sigma(f(x))) - (1-y)\log(1-\sigma(f(x)))$ where $\sigma(x) = 1/(1+e^{-x})$ is the sigmoid function. Repeat Step (a).

(d) Comment on any differences between logistic and quadratic loss in terms of optimization and test/train accuracy.

4. **Task B:** Multiclass classification

   (a) Train a neural network classifier with cross-entropy loss, namely $l(y, f(x)) = -\log(\text{softmax}(f(x))_y)$. Report the final test accuracy for the following choices

   - h=5
   - h=40
   - h=200

   Comment on the role of hidden units h on accuracy and the ease of optimization and test/train accuracy.

Solution

1) Here is the code used to normalize the data and initialize W,V matrices:

```
def format_data(data):
    flattened_mnist = data.astype(np.float32).reshape(-1, 784)
    mean = np.mean(flattened_mnist, axis=0)
    std = np.std(flattened_mnist, axis=0)  + 1e-8

    normalized_mnist = (flattened_mnist - mean) / std
    bias = np.ones((normalized_mnist.shape[0], 1), dtype=np.float32)
    return np.hstack((normalized_mnist, bias))

def init_matrix(h, d, K):
    # W \in R(h x d), V \in R(K x h)
    W_0 = np.random.randn(h, d) * np.sqrt(2.0 / d)
    V_0 = np.random.randn(K, h) * np.sqrt(2.0 / h)
    return W_0, V_0


mnist_train = datasets.MNIST(root='./data', train=True, download=True,
transform=transforms.ToTensor())
mnist_test = datasets.MNIST(root='./data', train=False, download=True,
transform=transforms.ToTensor())

x_train = format_data(mnist_train.data.numpy())
y_train = mnist_train.targets.numpy()
```

```
x_test = format_data(mnist_test.data.numpy())
y_test = mnist_test.targets.numpy()
```

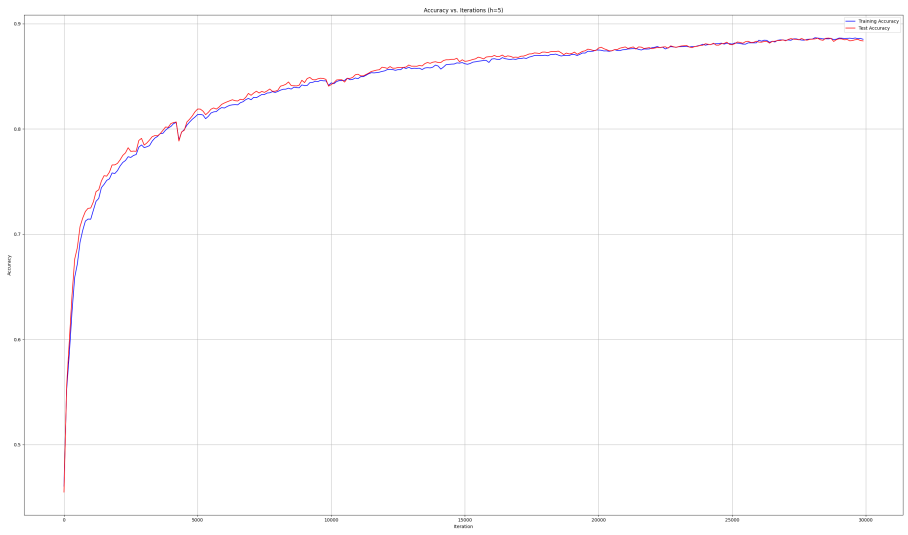
2) If we set $V_0 = W_0 = 0$ instead of random gaussian data, we will get stuck in at W_0, V_0. This is because in the model of $f(x; V, W) = V\text{ReLU}(Wx)$, Wx will always equal 0, due to the initial value. This leads to $f(x) = V(0) = 0$. This also causes all gradients to vanish as now $\nabla W, \nabla V = 0$, and the SGD steps will now get stuck at: $V_{t+1} = V_t - \eta\nabla V = 0 - \eta(0) = 0$ and $W_{t+1} = W_t - \eta\nabla W = 0 - \eta(0) = 0$. The model will fail to update ever, and thus is stuck.

3) a) Here is the function used to compute the gradient $\nabla f(V, W)$:

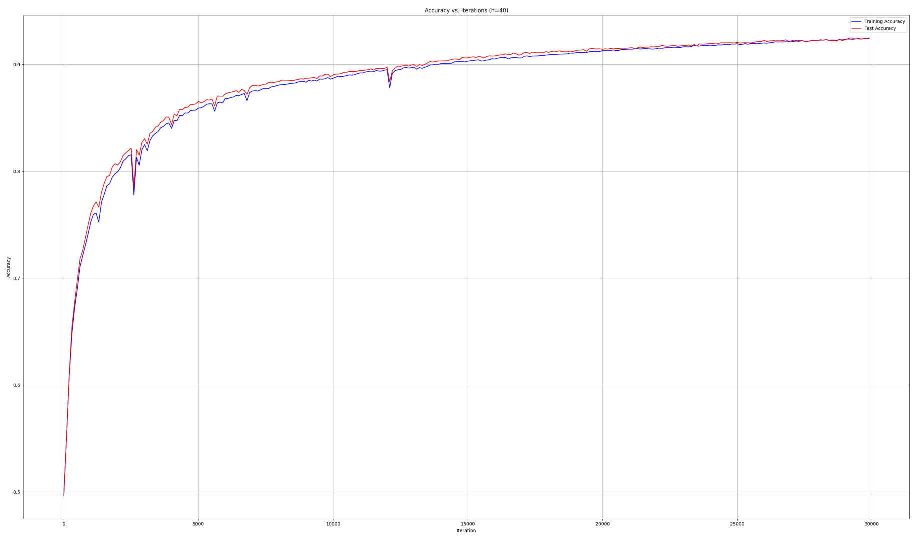
```
def compute_gradient(x, y, W, V, loss_type=''):
    # x : input data
    # y : label
    # W ; weight matrix for hidden layer (h x d)
    # V : weight matrix for output layer (K x h)
    # loss type : quadratic or logistic
    # return dV, dW

    size = x.shape[0]
    z = np.maximum(0, np.dot(x, W.T)) #ReLU(Wx)
    y_pred = np.dot(z, V.T)

    if(loss_type == 'quadratic'):
        # L = 1/2 * ||y - y_pred||^2
        dL = y_pred - y.reshape(-1,1)
    elif(loss_type == 'logistic'):
        # L = -y * log(sigma*f(x))-(1-y)log(1-sigma(f(x)))
        sigmoid = 1/(1+np.exp(-y_pred))
        dL = sigmoid - y.reshape(-1,1)

    #back prop
    dV = np.dot(dL.T, z) / size
    dZ = np.dot(dL, V)
    relu_deriv = (z > 0).astype(float)
    dZ = dZ * relu_deriv
    dW = np.dot(dZ.T, x) / size
    return dW, dV
```

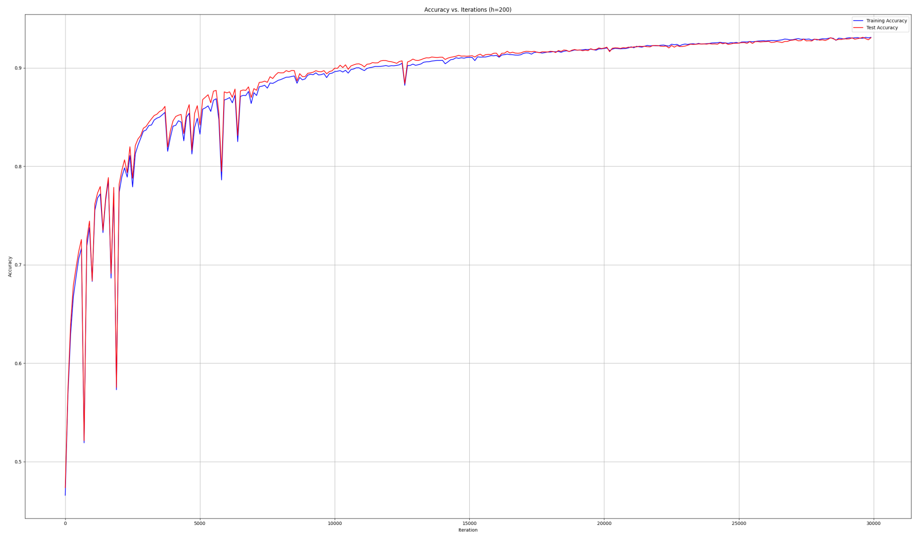
b) After training using quadratic loss, we achieve a final test accuracy of 0.8837 when h=5, 0.9247 when h=40, and 0.9306 when h=200. As we increase the number of hidden units, we marginally increase the training/testing accuracy. At a certain point increasing h does not greatly affect the accuracy results, as incresing h by a factor of 5 from h=40 to h=200 only led to a 0.5% increase in accuracy. This comes at the cost of a much greater time to calculate each iteration. Train, evaluate accuracy, and plot code:

```
# task a: binary classification
def train(x_train, y_train, x_test, y_test, h, loss_type='',
    learning_rate = 0.001, batch_size = 16, epochs = 8, report_freq = 100):

    d = x_train.shape[1]
    W, V = init_matrix(h, d, 1)
    train_accuracy_list = []
    test_accuracy_list = []
    inters = []

    n_iter = int((x_train.shape[0] / batch_size) * epochs)

    y_train_bin = (y_train > 4).astype(int)
    y_test_bin = (y_test > 4).astype(int)

    for iter in range(n_iter):
        #sample mini-batch
        idx = np.random.choice(x_train.shape[0], batch_size, replace=False)
        x_batch = x_train[idx]
        y_batch = y_train_bin[idx]

        #gradients
        dW, dV = compute_gradient(x_batch, y_batch, W, V, loss_type)

        #update
        W -= learning_rate * dW
        V -= learning_rate * dV

        if iter % report_freq == 0 or n_iter == -1 :
            train_acc = eval_acc(x_train, y_train_bin, W, V)
            test_acc = eval_acc(x_test, y_test_bin, W, V)

            train_accuracy_list.append(train_acc)
            test_accuracy_list.append(test_acc)
            inters.append(iter)

            print(f"Iteration {iter}: Train Accuracy: = {train_acc:.4f}, Test
            Accuracy: {test_acc:.4f}")

    return W, V, train_accuracy_list, test_accuracy_list, inters


def eval_acc(x, y, W, V):
    # x : input data
    # y : label
    # W ; weight matrix for hidden layer (h x d)
    # V : weight matrix for output layer (K x h)

    z = np.maximum(0, np.dot(x, W.T)) #ReLU(Wx)
    y_pred = np.dot(z, V.T)
    predictions = (y_pred > 0.5).astype(int).flatten()
    accuracy = np.mean(predictions == y)
    return accuracy


def plot_acc(train_acc, test_acc, inters, h):
    #plot train/test acc
    plt.figure(figsize=(10, 6))
    plt.plot(inters, train_acc, 'b-', label='Training Accuracy')
    plt.plot(inters, test_acc, 'r-', label='Test Accuracy')
    plt.title(f'Accuracy vs. Iterations (h={h})')
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.grid(True)
    plt.show()

mnist_train = datasets.MNIST(root='./data', train=True, download=True,
transform=transforms.ToTensor())
mnist_test = datasets.MNIST(root='./data', train=False, download=True,
transform=transforms.ToTensor())

x_train = format_data(mnist_train.data.numpy())
y_train = mnist_train.targets.numpy()
x_test = format_data(mnist_test.data.numpy())
y_test = mnist_test.targets.numpy()


print(f"Training using quadratic loss function")
for h in [5, 40, 200]:
    print(f"Training with hidden layer size: {h}")
    W, V, train_acc, test_acc, inters =
    train(x_train,y_train,x_test,y_test,h=h,loss_type='quadratic',report_freq=100)

    plot_acc(train_acc, test_acc, inters, h)
    print(f"Final test acc with h = {h}: {test_acc[-1]:.4f}")
```



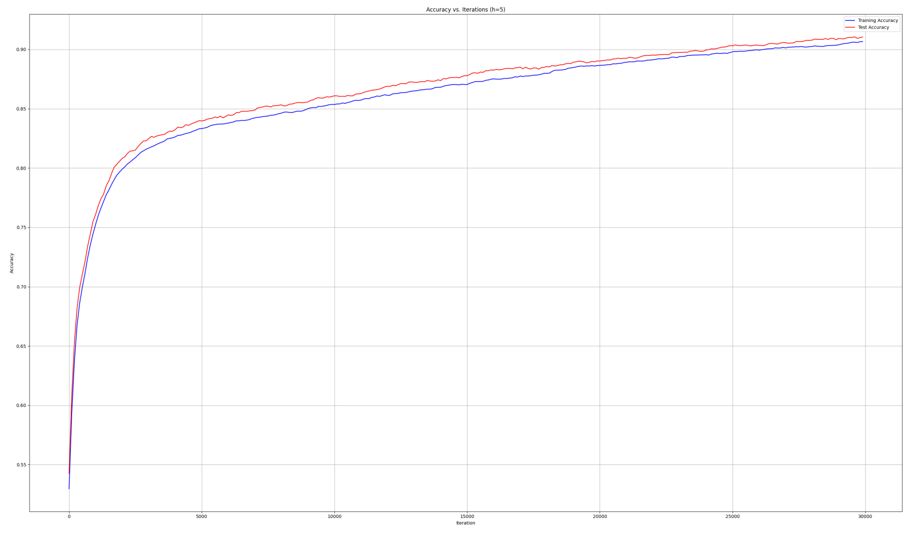
Here are the graphs produced:

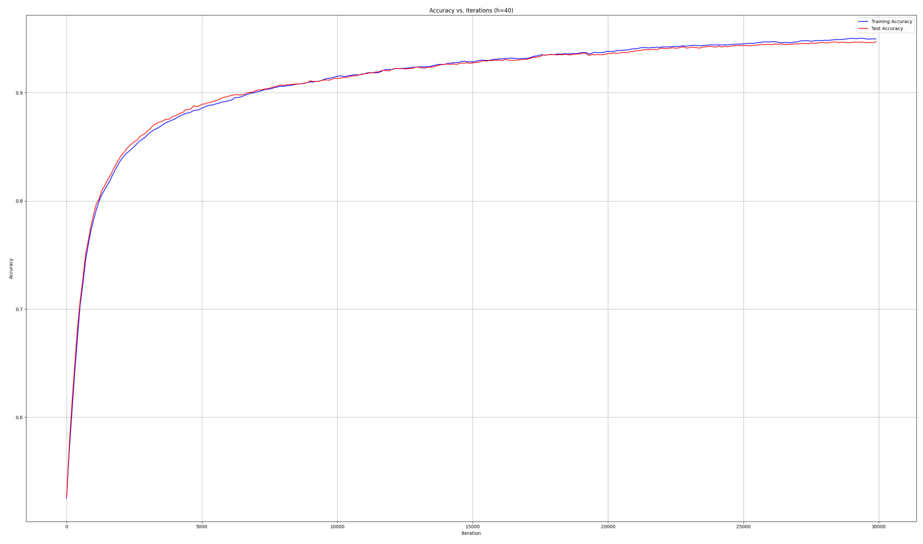

Accuracy vs. Iterations (h=40)
Training Accuracy
Test Accuracy
Accuracy
Iteration
0.5
0.6
0.7
0.8
0.9
0
5000
10000
15000
20000
25000
30000


Accuracy vs. Iterations (h=200)
Training Accuracy
Test Accuracy
Accuracy
Iteration
0.5
0.6
0.7
0.8
0.9
0
5000
10000
15000
20000
25000
30000

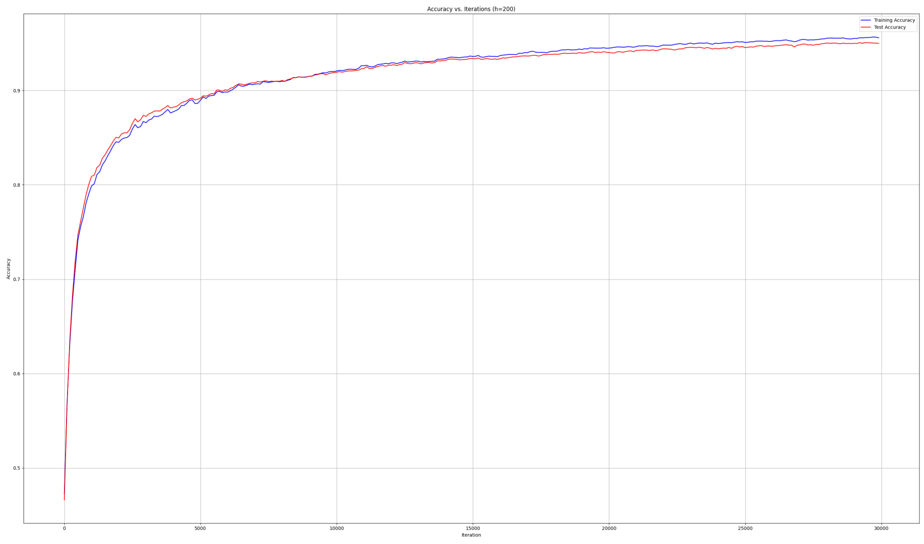
c) After training using logistic loss, we achieve a final test accuracy of 0.9105 when h=5, 0.9472 when h=40, and 0.9499 when h=200. Similar to using quadratic loss, we do not see significant accuracy improvements after a certain point, as increasing h=40 to h=200 only leads to a 0.3% increase in performance, worse than the improvement seen using quadratic loss. As we increase the number of hidden units, we marginally increase the training/testing accuracy. This comes at the cost of a much greater time to calculate each iteration. Changed gradient code:

```
def compute_gradient(x, y, W, V, loss_type=''):
    # x : input data
    # y : label
    # W ; weight matrix for hidden layer (h x d)
    # V : weight matrix for output layer (K x h)
    # loss type : quadratic or logistic
    # return dV, dW

    size = x.shape[0]
    z = np.maximum(0, np.dot(x, W.T)) #ReLU(Wx)
    y_pred = np.dot(z, V.T)

    if(loss_type == 'quadratic'):
        # L = 1/2 * ||y - y_pred||^2
        dL = y_pred - y.reshape(-1,1)
    elif(loss_type == 'logistic'):
        # L = -y * log(sigma*f(x))-(1-y)log(1-sigma(f(x)))
        sigmoid = 1/(1+np.exp(-y_pred))
        dL = sigmoid - y.reshape(-1,1)

    #back prop
    dV = np.dot(dL.T, z) / size
    dZ = np.dot(dL, V)
    relu_deriv = (z > 0).astype(float)
    dZ = dZ * relu_deriv
    dW = np.dot(dZ.T, x) / size
    return dW, dV

print(f"Training using logistic loss function")
for h in [5, 40, 200]:
    print(f"Training with hidden layer size: {h}")
    W, V, train_acc, test_acc, inters =
    train(x_train,y_train,x_test,y_test,h=h,loss_type='logistic',report_freq=100)

    plot_acc(train_acc, test_acc, inters, h)
    print(f"Final test acc with h = {h}: {test_acc[-1]:.4f}")
```





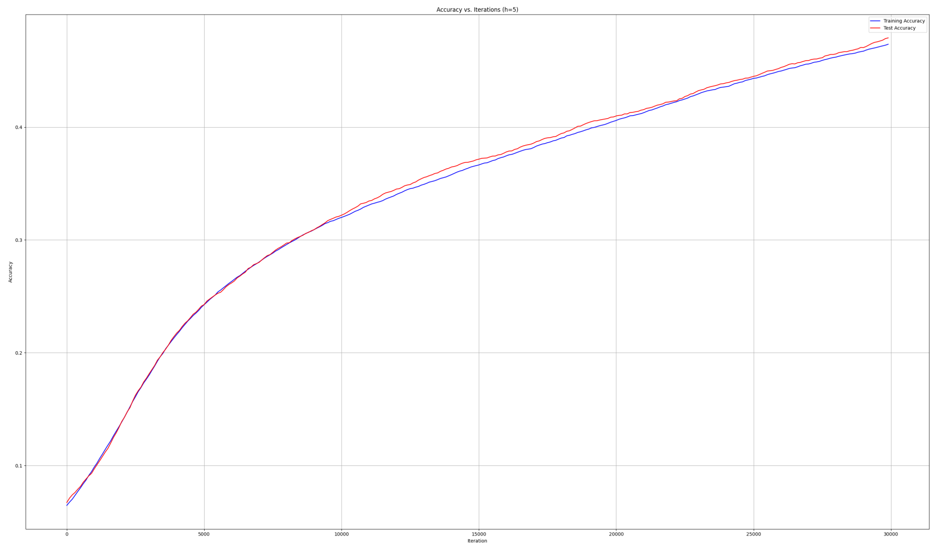
Here are the graphs produced:

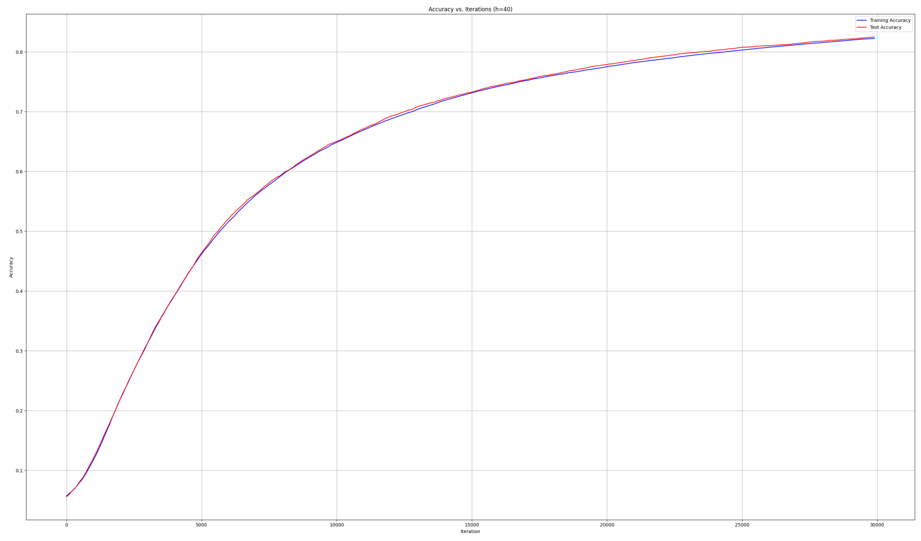


d) We find that logistic loss is a better loss function in terms of accuracy for this neural network, as we find it to be around $2-3\%$ more accurate for each h value compared to using quadratic loss. Additionally, we do not observe huge spikes in accuracy during the early stages of the training process as we see while using quadratic loss. Finally, we notice that using logistic loss leads to worse returns as we increase h after a certain value as we only get a 0.3% increase in accuracy after going from h=40 to h=200, compared to in quadratic loss of 0.5%.

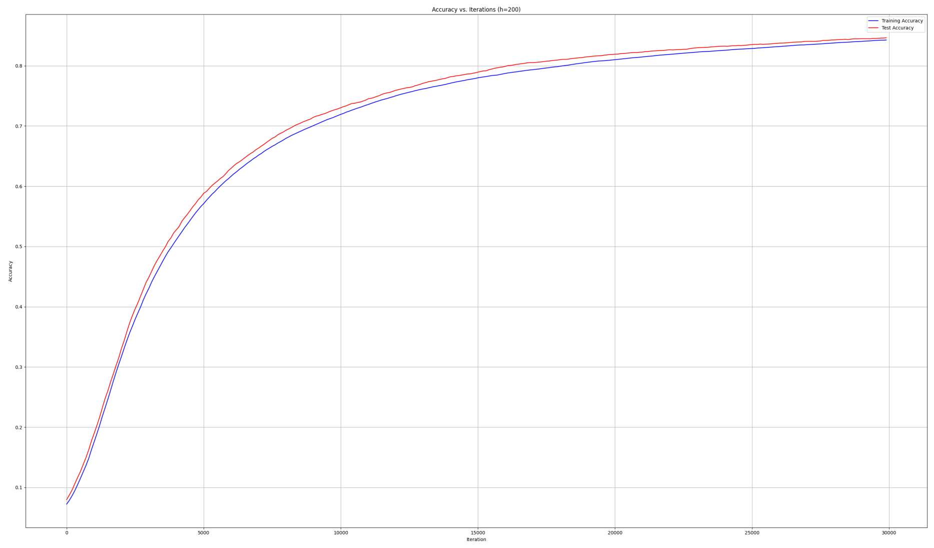
4) After training for multiclass classification, we achieve a final test accuracy of 0.4791 when h=5, 0.8249 when h=40, and 0.8465 when h=200. As we increase the number of hidden units, we drastically increase the training/testing accuracy. This is because smaller models tend to struggle in multiclass classification such as in Task B. This comes at the cost of a much greater time to calculate each iteration, similar to before. Additionally, we see diminishing returns of increasing the model size, as a five time increase from h=40 to h=200 only led to a 2% increase in accuracy, compared to the almost 35% increase in accuracy when going from h=5 to h=40. Updated multiclass code:

```
def one_hot_encode(y, num_classes=10):
    return np.eye(num_classes)[y.astype(int)]

def compute_multiclass_gradient(x, y, W, V):
    # l(y,f(x)) = - log(softmax(f(x))_y)

    # x : input data
    # y : label
    # W ; weight matrix for hidden layer (h x d)
    # V : weight matrix for output layer (K x h)
    # loss type : quadratic or logistic
    # return dV, dW

    size = x.shape[0]
    z = np.maximum(0, np.dot(x, W.T)) #ReLU(Wx)
    logits = np.dot(z, V.T)

    exps = np.exp(logits - np.max(logits, axis=1, keepdims=True))
    probs = exps / np.sum(exps, axis=1, keepdims=True)
    dL = (probs - y)/size

    #back prop
    dV = np.dot(dL.T, z) / size
    dZ = np.dot(dL, V)
    relu_deriv = (z > 0).astype(float)
    dZ = dZ * relu_deriv
    dW = np.dot(dZ.T, x) / size
    return dW, dV

def train_multiclass(x_train, y_train, x_test, y_test, h,
    learning_rate=0.001, batch_size=16, epochs=8, report_freq=100):

    d = x_train.shape[1]
    W, V = init_matrix(h, d, 10)
    train_accuracy_list = []
    test_accuracy_list = []
    inters = []

    n_iter = int((x_train.shape[0] / batch_size) * epochs)

    y_train_multi = one_hot_encode(y_train, 10)
    y_test_multi = one_hot_encode(y_test, 10)

    for iter in range(n_iter):
        #sample mini-batch
        idx = np.random.choice(x_train.shape[0], batch_size, replace=False)
        x_batch = x_train[idx]
        y_batch = y_train_multi[idx]

        #gradients
        dW, dV = compute_multiclass_gradient(x_batch, y_batch, W, V)

        #update
        W -= learning_rate * dW
        V -= learning_rate * dV
        if iter % report_freq == 0 or n_iter == -1 :
            train_acc = eval_multiclass_acc(x_train, y_train, W, V)
            test_acc = eval_multiclass_acc(x_test, y_test, W, V)

            train_accuracy_list.append(train_acc)
            test_accuracy_list.append(test_acc)
            inters.append(iter)

            print(f"Iteration {iter}: Train Accuracy: = {train_acc:.4f},
            Test Accuracy: {test_acc:.4f}")

    return W, V, train_accuracy_list, test_accuracy_list, inters

def eval_multiclass_acc(x,y,W,V):
    # x : input data
    # y : label
    # W ; weight matrix for hidden layer (h x d)
    # V : weight matrix for output layer (K x h)

    z = np.maximum(0, np.dot(x, W.T)) #ReLU(Wx)
    y_pred = np.dot(z, V.T)
    predictions = np.argmax(y_pred, axis=1)
    accuracy = np.mean(predictions == y)
    return accuracy

print(f"Multiclass Classification")
for h in [5, 40, 200]:
    print(f"Training with hidden layer size: {h}")
    W, V, train_acc, test_acc, inters = train_multiclass(x_train,y_train,x_test,y_test,h

    plot_acc(train_acc, test_acc, inters, h)
    print(f"Final test acc with h = {h}: {test_acc[-1]:.4f}")
```





Here are the graphs produced:



Complete Code:

```
import numpy as np
import matplotlib.pyplot as plt
from torchvision import datasets, transforms

def format_data(data):
    flattened_mnist = data.astype(np.float32).reshape(-1, 784)
    mean = np.mean(flattened_mnist, axis=0)
    std = np.std(flattened_mnist, axis=0)  + 1e-8

    normalized_mnist = (flattened_mnist - mean) / std
    bias = np.ones((normalized_mnist.shape[0], 1), dtype=np.float32)
    return np.hstack((normalized_mnist, bias))

def init_matrix(h, d, K):
    # W \in R(h x d), V \in R(K x h)
    W_0 = np.random.randn(h, d) * np.sqrt(2.0 / d)
    V_0 = np.random.randn(K, h) * np.sqrt(2.0 / h)
    return W_0, V_0

def one_hot_encode(y, num_classes=10):
    return np.eye(num_classes)[y.astype(int)]

def compute_gradient(x, y, W, V, loss_type=''):
    # x : input data
    # y : label
    # W ; weight matrix for hidden layer (h x d)
    # V : weight matrix for output layer (K x h)
    # loss type : quadratic or logistic
    # return dV, dW

    size = x.shape[0]
    z = np.maximum(0, np.dot(x, W.T)) #ReLU(Wx)
    y_pred = np.dot(z, V.T)

    if(loss_type == 'quadratic'):
        # L = 1/2 * ||y - y_pred||^2
        dL = y_pred - y.reshape(-1,1)
    elif(loss_type == 'logistic'):
        # L = -y * log(sigma*f(x))-(1-y)log(1-sigma(f(x)))
        sigmoid = 1/(1+np.exp(-y_pred))
        dL = sigmoid - y.reshape(-1,1)

    #back prop
    dV = np.dot(dL.T, z) / size
    dZ = np.dot(dL, V)
    relu_deriv = (z > 0).astype(float)
    dZ = dZ * relu_deriv
    dW = np.dot(dZ.T, x) / size
    return dW, dV


def compute_multiclass_gradient(x, y, W, V):
    # l(y,f(x)) = - log(softmax(f(x))_y)

    # x : input data
    # y : label
    # W ; weight matrix for hidden layer (h x d)
    # V : weight matrix for output layer (K x h)
    # loss type : quadratic or logistic
    # return dV, dW

    size = x.shape[0]
    z = np.maximum(0, np.dot(x, W.T)) #ReLU(Wx)
    logits = np.dot(z, V.T)

    exps = np.exp(logits - np.max(logits, axis=1, keepdims=True))
    probs = exps / np.sum(exps, axis=1, keepdims=True)
    dL = (probs - y)/size

    #back prop
    dV = np.dot(dL.T, z) / size
    dZ = np.dot(dL, V)
    relu_deriv = (z > 0).astype(float)
    dZ = dZ * relu_deriv
    dW = np.dot(dZ.T, x) / size
    return dW, dV


# task a: binary classification
def train(x_train, y_train, x_test, y_test, h, loss_type='',
    learning_rate = 0.001, batch_size = 16, epochs = 8, report_freq = 100):

    d = x_train.shape[1]
    W, V = init_matrix(h, d, 1)
    train_accuracy_list = []
    test_accuracy_list = []
    inters = []

    n_iter = int((x_train.shape[0] / batch_size) * epochs)

    y_train_bin = (y_train > 4).astype(int)
    y_test_bin = (y_test > 4).astype(int)

    for iter in range(n_iter):
        #sample mini-batch
        idx = np.random.choice(x_train.shape[0], batch_size, replace=False)
        x_batch = x_train[idx]
        y_batch = y_train_bin[idx]

        #gradients
        dW, dV = compute_gradient(x_batch, y_batch, W, V, loss_type)

        #update
        W -= learning_rate * dW
        V -= learning_rate * dV

        if iter % report_freq == 0 or n_iter == -1 :
            train_acc = eval_acc(x_train, y_train_bin, W, V)
            test_acc = eval_acc(x_test, y_test_bin, W, V)

            train_accuracy_list.append(train_acc)
            test_accuracy_list.append(test_acc)
            inters.append(iter)

            print(f"Iteration {iter}: Train Accuracy: = {train_acc:.4f},
            Test Accuracy: {test_acc:.4f}")

    return W, V, train_accuracy_list, test_accuracy_list, inters

def train_multiclass(x_train, y_train, x_test, y_test, h,
    learning_rate=0.001, batch_size=16, epochs=8, report_freq=100):

    d = x_train.shape[1]
    W, V = init_matrix(h, d, 10)
    train_accuracy_list = []
    test_accuracy_list = []
    inters = []

    n_iter = int((x_train.shape[0] / batch_size) * epochs)

    y_train_multi = one_hot_encode(y_train, 10)
    y_test_multi = one_hot_encode(y_test, 10)

    for iter in range(n_iter):
        #sample mini-batch
        idx = np.random.choice(x_train.shape[0], batch_size, replace=False)
        x_batch = x_train[idx]
        y_batch = y_train_multi[idx]

        #gradients
        dW, dV = compute_multiclass_gradient(x_batch, y_batch, W, V)

        #update
        W -= learning_rate * dW
        V -= learning_rate * dV

        if iter % report_freq == 0 or n_iter == -1 :
            train_acc = eval_multiclass_acc(x_train, y_train, W, V)
            test_acc = eval_multiclass_acc(x_test, y_test, W, V)

            train_accuracy_list.append(train_acc)
            test_accuracy_list.append(test_acc)
            inters.append(iter)

            print(f"Iteration {iter}: Train Accuracy: = {train_acc:.4f},
            Test Accuracy: {test_acc:.4f}")

    return W, V, train_accuracy_list, test_accuracy_list, inters
```

```
def eval_acc(x, y, W, V):
    # x : input data
    # y : label
    # W ; weight matrix for hidden layer (h x d)
    # V : weight matrix for output layer (K x h)

    z = np.maximum(0, np.dot(x, W.T)) #ReLU(Wx)
    y_pred = np.dot(z, V.T)
    predictions = (y_pred > 0.5).astype(int).flatten()
    accuracy = np.mean(predictions == y)
    return accuracy

def eval_multiclass_acc(x,y,W,V):
    # x : input data
    # y : label
    # W ; weight matrix for hidden layer (h x d)
    # V : weight matrix for output layer (K x h)

    z = np.maximum(0, np.dot(x, W.T)) #ReLU(Wx)
    y_pred = np.dot(z, V.T)
    predictions = np.argmax(y_pred, axis=1)
    accuracy = np.mean(predictions == y)
    return accuracy


def plot_acc(train_acc, test_acc, inters, h):
    #plot train/test acc
    plt.figure(figsize=(10, 6))
    plt.plot(inters, train_acc, 'b-', label='Training Accuracy')
    plt.plot(inters, test_acc, 'r-', label='Test Accuracy')
    plt.title(f'Accuracy vs. Iterations (h={h})')
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.grid(True)
    plt.show()

mnist_train = datasets.MNIST(root='./data', train=True, download=True,
transform=transforms.ToTensor())
mnist_test = datasets.MNIST(root='./data', train=False, download=True,
transform=transforms.ToTensor())

x_train = format_data(mnist_train.data.numpy())
y_train = mnist_train.targets.numpy()
```

```
x_test = format_data(mnist_test.data.numpy())
y_test = mnist_test.targets.numpy()


print(f"Training using quadratic loss function")
for h in [5, 40, 200]:
    print(f"Training with hidden layer size: {h}")
    W, V, train_acc, test_acc, inters =
    train(x_train,y_train,x_test,y_test,h=h,loss_type='quadratic',report_f
    req=100)

    plot_acc(train_acc, test_acc, inters, h)
    print(f"Final test acc with h = {h}: {test_acc[-1]:.4f}")


print(f"Training using logistic loss function")
for h in [5, 40, 200]:
    print(f"Training with hidden layer size: {h}")
    W, V, train_acc, test_acc, inters =
    train(x_train,y_train,x_test,y_test,h=h,loss_type='logistic',report_fr
    eq=100)

    plot_acc(train_acc, test_acc, inters, h)
    print(f"Final test acc with h = {h}: {test_acc[-1]:.4f}")

print(f"Multiclass Classification")
for h in [5, 40, 200]:
    print(f"Training with hidden layer size: {h}")
    W, V, train_acc, test_acc, inters =
    train_multiclass(x_train,y_train,x_test,y_test,h=h,report_freq=100)

    plot_acc(train_acc, test_acc, inters, h)
    print(f"Final test acc with h = {h}: {test_acc[-1]:.4f}")
```