Homework #3
EE 541: Fall 2023

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

1) An MLP has two input nodes, one hidden layer, and two outputs. Recall that the output for layer l is given by a(l) = hl□Wla(l−1) + bl. The two sets of weights and biases are given by:
W1 = 1 −2 3 4
b1 =1 0
W2 = 2 2 2 −3
b2 = 0 −4
The non-linear activation for the hidden layer is ReLU (rectified linear unit) – that is h(x) = max (x, 0). The output layer is linear (i.e., identity activation function). What is the output activation for input x = [+1;−1]T ?

Output: y = [8 4]

APPENDIX:

```
import numpy as np            #define numpy library

class MLP:                    #define class
    def __init__(y):          #define init function
        # Define model parameters
        y.weights = [np.array([[1, -2], [3, 4]]), np.array([[2, 2], [2,
-3]])]          #weights
        y.biases = [np.array([1, 0]), np.array([0, -4])]
#base

    def predict(y, x):       #define predict
        # Forward pass through the network
        for w, b in zip(y.weights, y.biases):
#zip to take values as a pair
            x = y.relu(np.dot(w, x) + b)
        return x

    def relu(y, z):           #define relu
```

```python
        # ReLU activation function
        return np.maximum(z, 0)

# Input
x = np.array([1, -1])

# Create the model
model = MLP()

# Make a prediction
y = model.predict(x)

# Print the output
print(f'Output: y = {y}')
```

2. The hd5 format can store multiple data objects in a single file each keyed by object name – e.g., you can store a numpy float array called regressor and a numpy integer array called labels in the same file. Hd5 also allows fast non-sequential access to objects without scanning the entire file. This means you can efficiently access objects and data such as x[idxs] with non-consecutive indexes e.g., idxs = [2, 234, 512]. This random-access property is useful when extracting a random subset from a larger training database. In this problem you will create an hd5 file containing a numpy array of binary random sequences that you generate yourself. Follow these steps:
(1) Run the provided template python file (random binary collection.py). The script is set to DEBUG mode by default.
(2) Experiment with the assert statements to trap errors and understand what they are doing by using the shape method on numpy arrays, etc.
(3) Set the DEBUG flag to False. Manually create 25 binary sequences each with length 20. It is important that you do this by hand, i.e., , do not use a coin, computer, or random number generator.
(4) Verify that your hd5 file was written properly by checking that it can be read-back.
(5) Submit your hd5 file as directed

APPENDIX:

```
## copyright, Keith Chugg
##  EE599, 2020

#################################################
## this is a template to illustrate hd5 files
##
## also can be used as template for HW1 problem
##################################################

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

DEBUG = False
DATA_FNAME = 'brandon_franzke_hw1_1.hd5'

if DEBUG:
    num_sequences = 3
    sequence_length = 4
else:
    num_sequences = 25
    sequence_length = 20

### Enter your data here...
### Be sure to generate the data by hand.  DO NOT:
###     copy-n-paste
###     use a random number generator
###
x_list = [
    [ 0, 1, 1, 0],
    [ 1, 1, 0, 0],
    [ 0, 0, 0, 1]
]

user_list = [
    [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
    [ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
    [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
```

```
    [ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
    [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
    [ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
    [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
    [ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
    [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
    [ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
    [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
    [ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
    [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
    [ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
    [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
    [ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
    [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
    [ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
    [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
    [ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
    [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
    [ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
    [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
    [ 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
    [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
]

# convert list to a numpy array...
human_binary = np.asarray(x_list)
user_binary = np.asarray(user_list)

### do some error trapping:

if DEBUG:
    assert human_binary.shape[0] == num_sequences, 'Error: the number of sequences was entered incorrectly'
    assert human_binary.shape[1] == sequence_length, 'Error: the length of the seqeunces is incorrect'
else:
    assert user_binary.shape[0] == num_sequences
    assert user_binary.shape[1] == sequence_length
```

```python
# the with statement opens the file, does the business, and close it up
for us...
with h5py.File(DATA_FNAME, 'w') as hf:
    hf.create_dataset('human_binary', data = human_binary)
    ## note you can write several data arrays into one hd5 file, just give
each a different name.
    hf.create_dataset('user_binary', data = user_binary)

# Let's read it back from the file and then check to make sure it is as we
wrote...
with h5py.File(DATA_FNAME, 'r') as hf:
    hb_copy = hf['human_binary'][:]
    user_copy = hf['user_binary'][:]

### this will throw and error if they are not the same...
if DEBUG:
    np.testing.assert_array_equal(human_binary, hb_copy)
else:
    np.testing.assert_array_equal(user_binary, user_copy)
```

3) Logistic regression
The MNIST dataset of handwritten digits is one of the earliest and most used datasets to benchmark machine learning classifiers. Each datapoint contains 784 input features – the pixel values from a 28 × 28 image – and belongs to one of 10 output classes – represented by the numbers 0-9. In this problem you will use numpy to classify input images using a logistic-regression. Use only Python standard library modules, numpy, and matplotlib for this problem.

(a) Logistic "2" detector
In this part you will use the provided MNIST handwritten-digit data to build and train a logistic "2" detector:
y = (1 x is a "2" 0 else.
A logistic classifier takes learned weight vector w = [w1,w2, . . .wL]T and the unregularized offset bias b ≜ w0 to estimate a probability that an input vector x = [x1, x2, . . . , xL]T is "2":
p(x) = P [Y = 1|x,w] = 1 1 + exp − PL i=1 wk · xk + w0 =11 + exp (−(wT x + w0)).
Train a logistic classifier to find weights that minimize the binary log-loss (also called the binary cross entropy loss):
l(w) = −1 N XN i=1 (yi log p(x)) + (1 − yi) log (1 − p(x))

where the sum is over the N samples in the training set. Train your model until convergence according to some metric you choose. Experiment with variations of ℓ1- and/or ℓ2-regularization to stabilize training and improve generalization.
Submit answers to the following.
i. How did you determine a learning rate? What values did you try? What was your final value?
ii. Describe the method you used to establish model convergence.
iii. What regularizers did you try? Specifically, how did each impact your model or improve its performance?
iv. Plot log-loss (i.e., learning curve) of the training set and test set on the same figure. On a separate figure plot the accuracy against iteration number of your model on the training set and test set. Plot each as a function of the iteration number.
v. Classify each input to the binary output “digit is a 2” using a 0.5 threshold. Compute the final loss and final accuracy for both your training set and test set.
Submit your trained weights to Autolab. Save your weights and bias to an hdf5 file. Use keys w and b for the weights and bias, respectively. w should be a length-784 numpy vector/array and b should be a numpy scalar. Use the following as guidance:
with h5py.File(outFile, 'w') as hf:
hf.create_dataset('w', data = np.asarray(weights))
hf.create_dataset('b', data = np.asarray(bias))
Note: you will not be scored on your models overall accuracy. But a low-score may indicate errors in training or poor optimization.

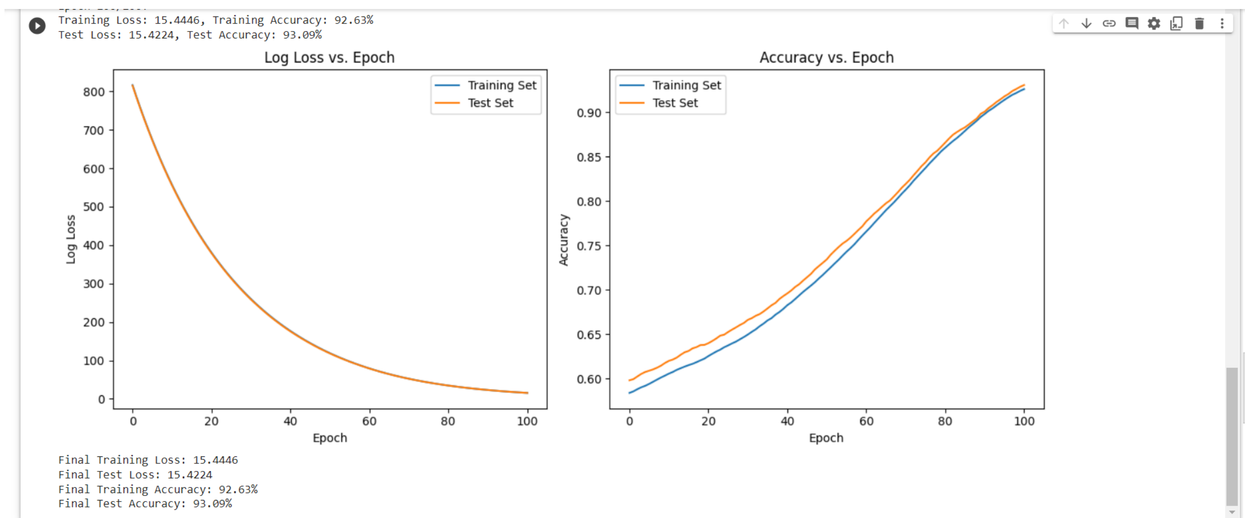
Answer:

i) Predefined Values: There are occasions when conventional learning rates work effectively for certain algorithms or designs. Depending on the dataset and task complexity, common beginning learning rates for logistic regression are 0.1, 0.01, or 0.001.

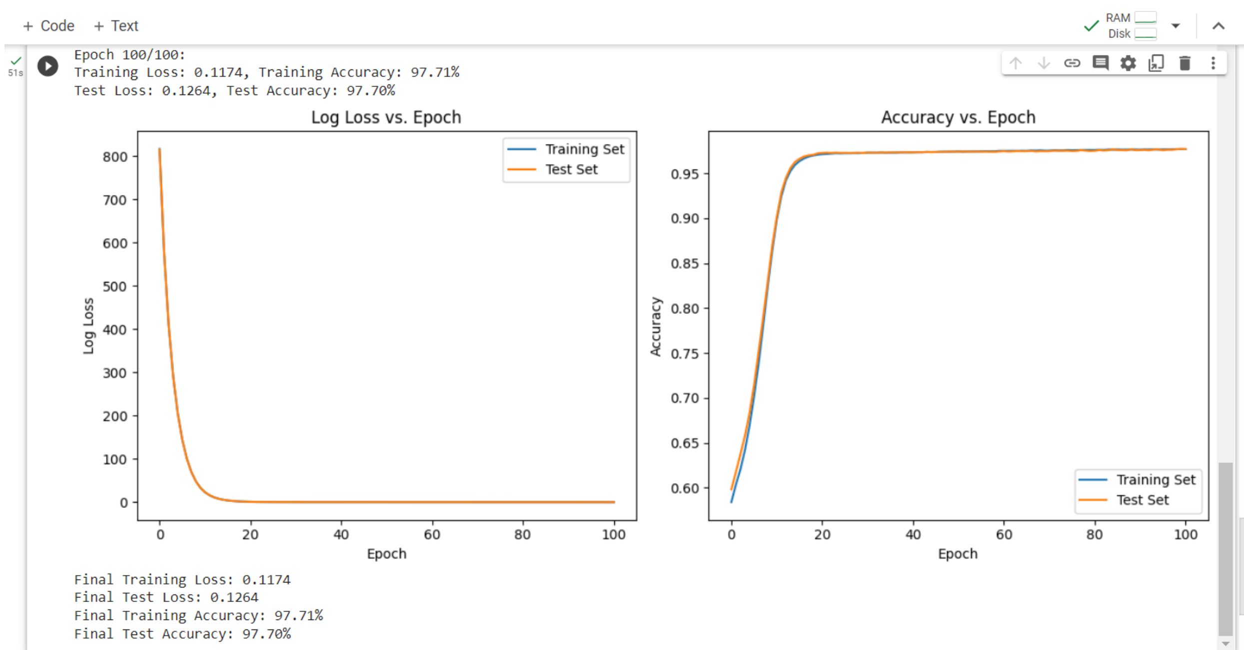
Experiment and iterate: This is frequently an iterative process. Begin with a modest learning rate, train your model, and monitor its performance. If it's converging too slowly or diverging too quickly, modify the learning rate and repeat the procedure.

Tried Learning rate :

Learning Rate: 0.001
Epochs: 100
Lamda: 0.01



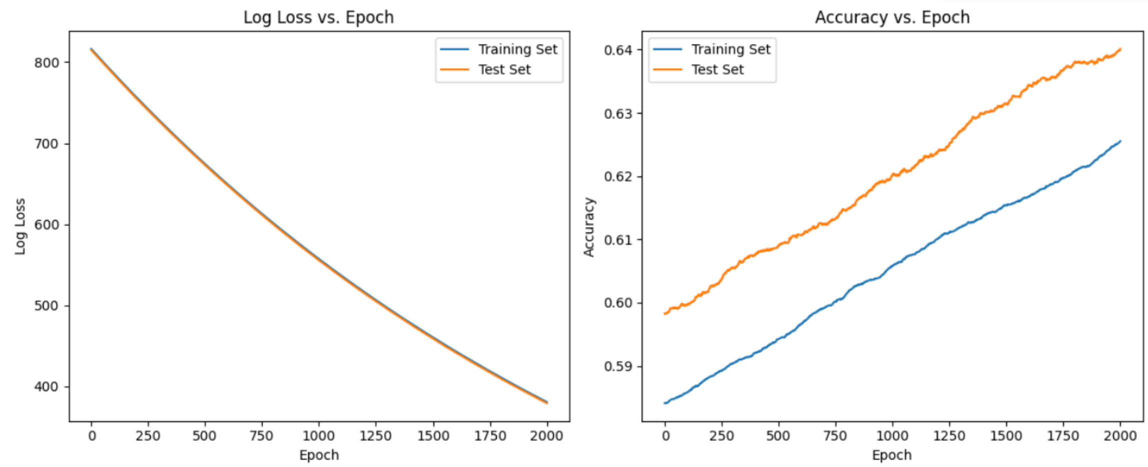
Learning Rate: 0.009
Epochs: 100
Lamda: 0.01



Learning Rate: 0.0001
Epochs: 2000
Lamda: 0.01

+ Code + Text

```
Training Loss: 380.5250, Training Accuracy: 62.55%
Test Loss: 379.0647, Test Accuracy: 64.01%
```

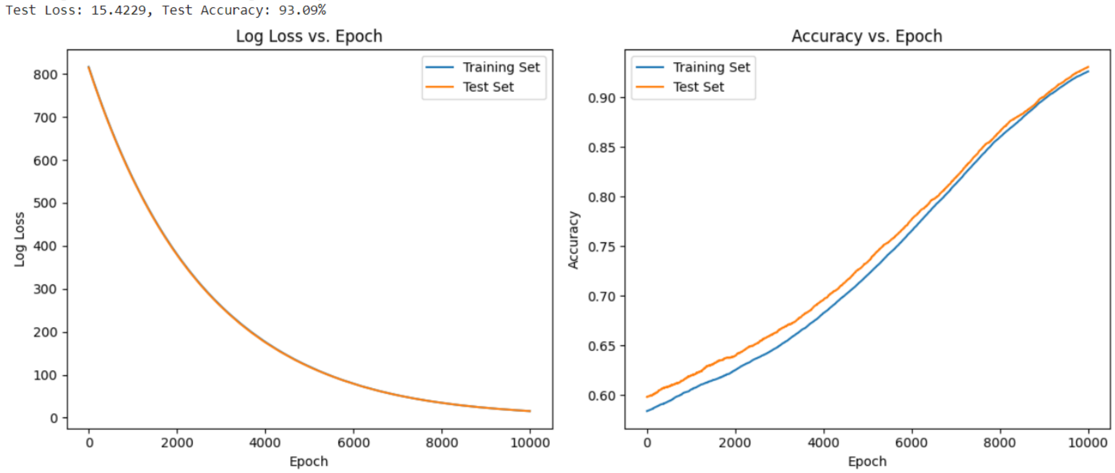



```
Final Training Loss: 380.5250
Final Test Loss: 379.0647
Final Training Accuracy: 62.55%
Final Test Accuracy: 64.01%
```

Learning Rate: 0.0001
Epochs: 10000
Lamda: 0.01

+ Code + Text

```
Epoch 10000/10000:
Training Loss: 15.4451, Training Accuracy: 92.63%
Test Loss: 15.4229, Test Accuracy: 93.09%
```




```
Final Training Loss: 15.4451
Final Test Loss: 15.4229
Final Training Accuracy: 92.63%
Final Test Accuracy: 93.09%
```

✓ 1h 20m 23s completed at 7:10 AM

FINAL LEARNING RATE: 0.009

Various accuracy and losses for the training and test datasets are achieved by varying the Learning rate using standard rates and adjusting the epoch value. Depending on the model, the trail and error approach is employed to achieve optimal accuracy. However, while raising the Epochs can yield better results in general, in the actual world, it increases the time required to create a model, making it unfeasible.

ii)

```
accuracy = np.sum((y_pred > 0.5).astype(int) == yt) / len(yt)
```

The line computes the predictive accuracy of the logistic regression model by comparing it to the ground truth labels. It translates projected probabilities into binary predictions using a 0.5 threshold, verifies for correctness, and calculates accuracy as the ratio of accurate predictions to total number of cases in the dataset. This accuracy parameter is used to track the model's convergence throughout training, with convergence usually occurring when accuracy stabilizes or plateaus, suggesting consistent and accurate predictions.

iii)

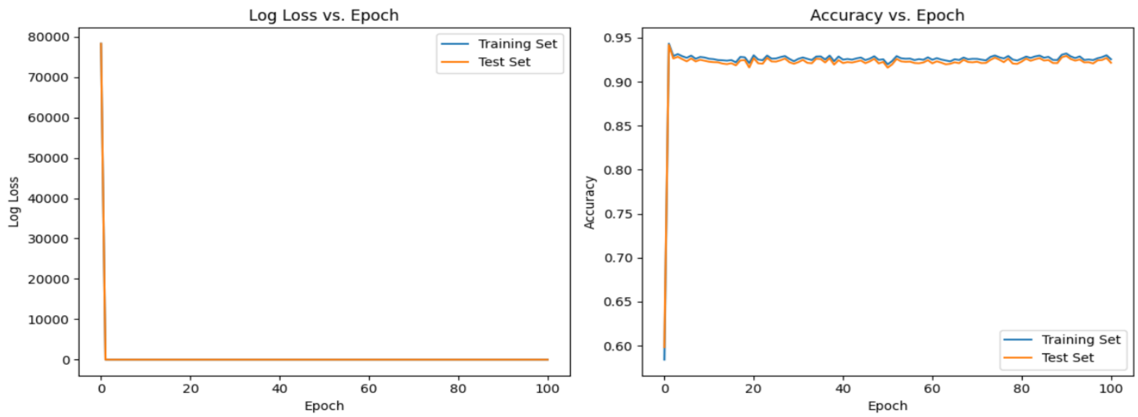
lam * np.sum(w**2)

L2 regularization promotes the model to use tiny weight values, which can aid in the prevention of overfitting. It basically adds a penalty term to the loss for having big weights, making the model more resilient and perhaps enhancing generalization to previously unknown data.

Similar to the Learning rate, by reducing the value of regularization strength the accuracy of the model gets increased. By trying various values an optimum value can be obtained but the overall time complexity of code gets increased which increases the time to train the model.

Learning Rate: 0.009
Epochs: 100
Lamda: 1

```
Epoch 100/100:
Training Loss: 0.2272, Training Accuracy: 92.56%
Test Loss: 0.2304, Test Accuracy: 92.16%
```

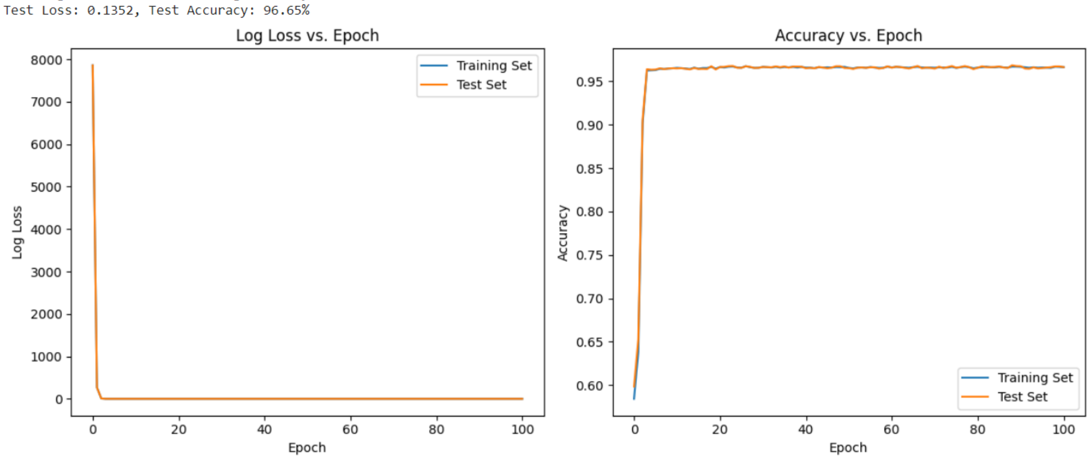



```
Final Training Loss: 0.2272
Final Test Loss: 0.2304
Final Training Accuracy: 92.56%
Final Test Accuracy: 92.16%
```

✓ 51s completed at 7:13 AM

Learning Rate: 0.009
Epochs: 100
Lamda: 0.1

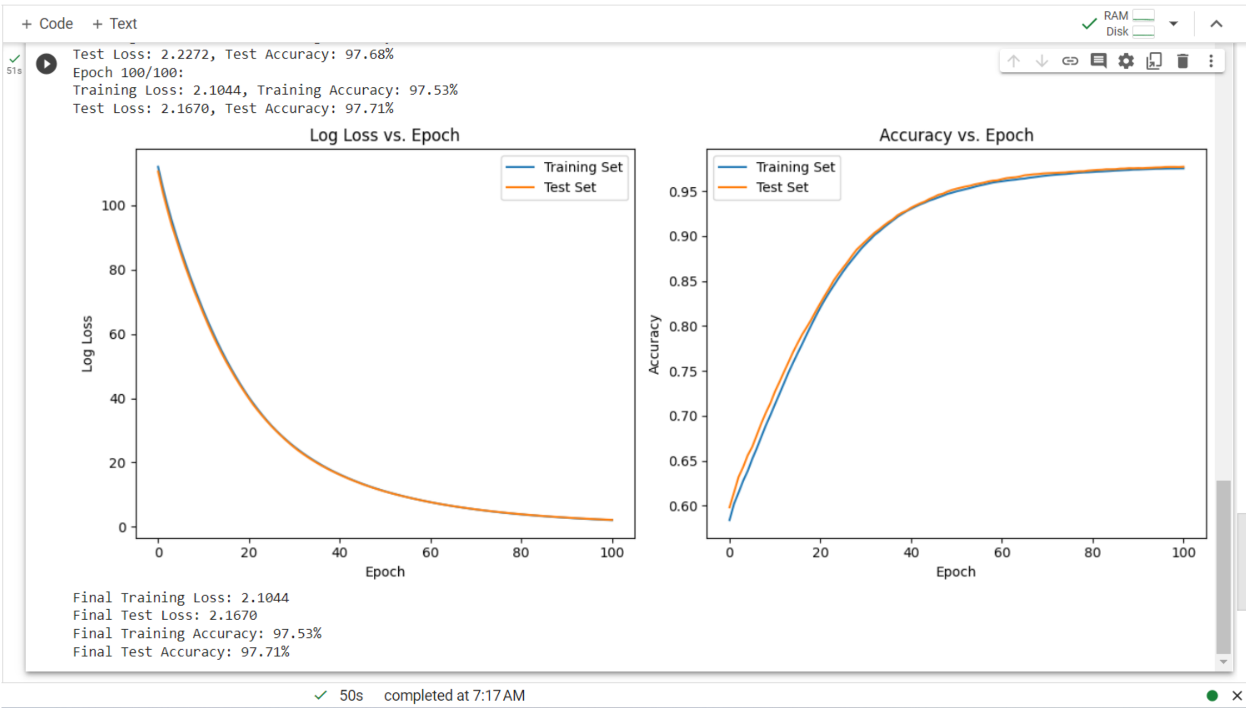
```
Test Loss: 0.1351, Test Accuracy: 96.73%
Epoch 100/100:
Training Loss: 0.1330, Training Accuracy: 96.65%
Test Loss: 0.1352, Test Accuracy: 96.65%
```




```
Final Training Loss: 0.1330
Final Test Loss: 0.1352
Final Training Accuracy: 96.65%
Final Test Accuracy: 96.65%
```

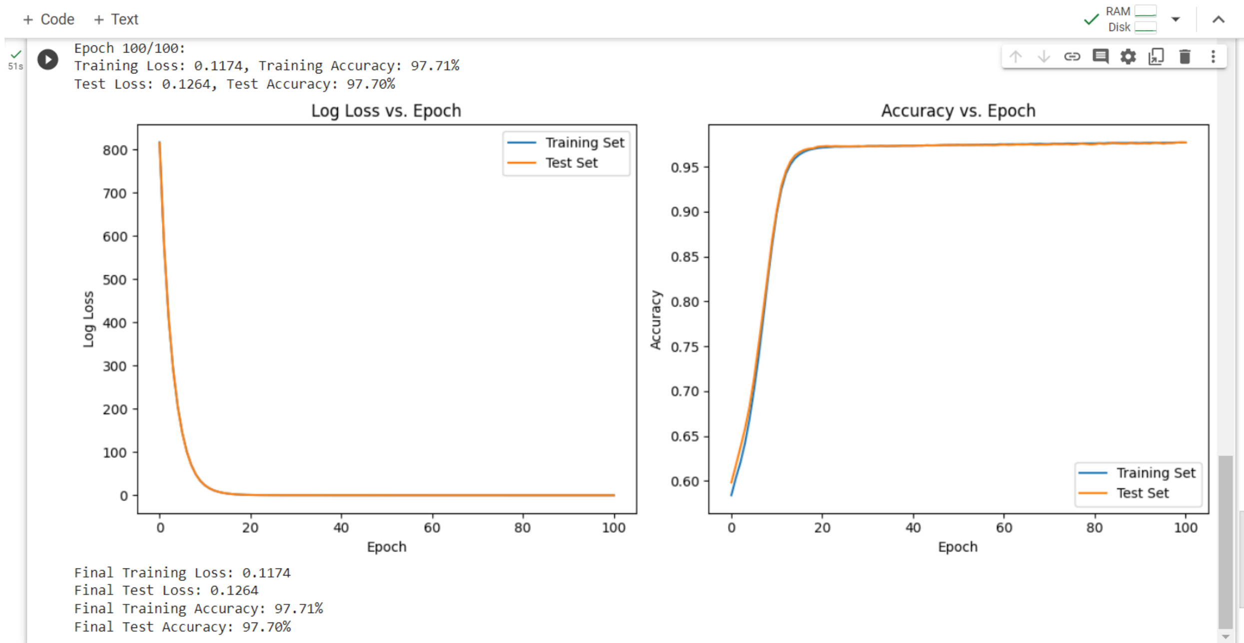
✓ 51s completed at 7:15 AM

Learning Rate: 0.009
Epochs: 100
Lamda: 0.001



iv)

Training Accuracy: 97.71%
Test Accuracy: 97.70%
Training Loss:0.1174
Test Loss:0.1264



v)

(y_pred > 0.5).astype(int)

Individual Losses:

```
final_train_loss, final_train_accuracy = calculate_loss_accuracy(xtrain,
ytrain_binary[:, 2], yt, w, b)
final_test_loss, final_test_accuracy = calculate_loss_accuracy(xtest,
ytest_binary[:, 2], ytesto, w, b)
```

APPENDIX:

```
# Import necessary libraries
import h5py
import numpy as np
import matplotlib.pyplot as plt

# Create a class for Logistic Regression Classifier
class LogisticRegressionClassifier:
    def __init__(self, learning_rate=0.001, n_epochs=100,
regularization_strength=0.01):
```

```
        # Initialize hyperparameters and variables
        self.learning_rate = learning_rate
        self.n_epochs = n_epochs
        self.regularization_strength = regularization_strength
        self.weights = None
        self.bias = None
        self.train_losses = []
        self.test_losses = []
        self.train_accuracies = []
        self.test_accuracies = []

    def fit(self, x_train, y_train, x_test, y_test):
        # Binary classification: 2 or not 2
        y_train_binary = (y_train == [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0]).astype(int)
        y_test_binary = (y_test == [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0]).astype(int)

        # Initialize weights and bias for logistic regression
        np.random.seed(seed=1)
        self.weights = np.random.normal(0, 10, size=(784,))
        self.bias = 0.0

        # Lists to store losses and accuracies during training
        self.train_losses = []
        self.test_losses = []
        self.train_accuracies = []
        self.test_accuracies = []

        # Function to calculate loss and accuracy
        def calculate_loss_accuracy(x, y, yt):
            scores = np.dot(x, self.weights) + self.bias
            y_pred = 1 / (1 + np.exp(-scores))

            loss = -np.sum((y * np.log(y_pred + 1e-40) + (1 - y) *
np.log(1 - y_pred + 1e-40))) / len(y) + self.regularization_strength *
np.sum(self.weights**2)
            accuracy = np.sum((y_pred > 0.5).astype(int) == yt) / len(yt)
            return loss, accuracy
```

```
        # Define 'yt' here for test data
        yt = y_train_binary[:, 2]

        # Define 'y_test_o' here for test data
        y_test_o = y_test_binary[:, 2]

        # Training loop
        for epoch in range(self.n_epochs):
            # Compute predicted probabilities for the entire training set
            scores = np.dot(x_train, self.weights) + self.bias
            y_pred = 1 / (1 + np.exp(-scores))

            # Compute gradients for the entire training set
            dw = (1 / len(yt)) * np.dot(x_train.T, (y_pred -
y_train_binary[:, 2])) + 2 * self.regularization_strength * self.weights
            db = (1 / len(yt)) * np.sum(y_pred - y_train_binary[:, 2])

            # Update weights and bias
            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

            # Calculate and store loss and accuracy for both training and
test sets
            train_loss, train_accuracy = calculate_loss_accuracy(x_train,
y_train_binary[:, 2], yt)
            test_loss, test_accuracy = calculate_loss_accuracy(x_test,
y_test_binary[:, 2], y_test_o)

            self.train_losses.append(train_loss)
            self.test_losses.append(test_loss)
            self.train_accuracies.append(train_accuracy)
            self.test_accuracies.append(test_accuracy)

            # Print epoch-wise information
            print(f"Epoch {epoch + 1}/{self.n_epochs}:")
            print(f"Training Loss: {train_loss:.4f}, Training Accuracy:
{train_accuracy:.2%}")
            print(f"Test Loss: {test_loss:.4f}, Test Accuracy:
{test_accuracy:.2%}")
```

```
    def plot_learning_curves(self):
        # Plot learning curves
        plt.figure(figsize=(12, 5))
        plt.subplot(1, 2, 1)
        plt.plot(self.train_losses, label='Training Set')
        plt.plot(self.test_losses, label='Test Set')
        plt.xlabel('Epoch')
        plt.ylabel('Log Loss')
        plt.title('Log Loss vs. Epoch')
        plt.legend()

        plt.subplot(1, 2, 2)
        plt.plot(self.train_accuracies, label='Training Set')
        plt.plot(self.test_accuracies, label='Test Set')
        plt.xlabel('Epoch')
        plt.ylabel('Accuracy')
        plt.title('Accuracy vs. Epoch')
        plt.legend()

        plt.tight_layout()
        plt.show()

    def get_final_metrics(self, x_train, y_train, x_test, y_test):
        # Binary classification: 2 or not 2
        y_train_binary = (y_train == [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0]).astype(int)
        y_test_binary = (y_test == [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0]).astype(int)

        # Define 'yt' here for test data
        yt = y_train_binary[:, 2]

        # Define 'y_test_o' here for test data
        y_test_o = y_test_binary[:, 2]

        # Calculate final loss and accuracy
        final_train_loss, final_train_accuracy =
self.calculate_loss_accuracy(x_train, y_train_binary[:, 2], yt)
        final_test_loss, final_test_accuracy =
self.calculate_loss_accuracy(x_test, y_test_binary[:, 2], y_test_o)
```

```
        return final_train_loss, final_test_loss, final_train_accuracy,
final_test_accuracy

    def calculate_loss_accuracy(self, x, y, yt):
        scores = np.dot(x, self.weights) + self.bias
        y_pred = 1 / (1 + np.exp(-scores))

        loss = -np.sum((y * np.log(y_pred + 1e-40) + (1 - y) * np.log(1 -
y_pred + 1e-40))) / len(y) + self.regularization_strength *
np.sum(self.weights**2)
        accuracy = np.sum((y_pred > 0.5).astype(int) == yt) / len(yt)
        return loss, accuracy

# Example usage of the class:
if __name__ == "__main__":
    # Load the training and test datasets
    train_data = h5py.File('mnist_traindata.hdf5', 'r')
    x_train = np.asarray(train_data['xdata'])
    y_train = np.asarray(train_data['ydata'])

    test_data = h5py.File('mnist_testdata.hdf5')
    x_test = np.asarray(test_data['xdata'])
    y_test = np.asarray(test_data['ydata'])

    # Create and train the LogisticRegressionClassifier
    clf = LogisticRegressionClassifier()
    clf.fit(x_train, y_train, x_test, y_test)

    # Plot learning curves
    clf.plot_learning_curves()

    # Get final metrics
    final_train_loss, final_test_loss, final_train_accuracy,
final_test_accuracy = clf.get_final_metrics(x_train, y_train, x_test,
y_test)

    print(f"Final Training Loss: {final_train_loss:.4f}")
    print(f"Final Test Loss: {final_test_loss:.4f}")
    print(f"Final Training Accuracy: {final_train_accuracy:.2%}")
```

```python
    print(f"Final Test Accuracy: {final_test_accuracy:.2%}")
```

(b) Softmax classification: gradient descent (GD)
In this part you will use soft-max to perform multi-class classification instead of distinct "one against all" detectors. The target vector
[Y]l =(1 x is an "l" 0 else. for l = 0, . . . ,K − 1. You can alternatively consider a scalar output Y equal to the value in {0, 1, . . . ,K − 1} corresponding to the class of input x. Construct a logistic classifier that uses K separate linear weight vectors w0,w1, . . . ,wK−1. Compute estimated probabilities for each class given input x and select the class with the largest score among your K predictors:
P [Y = l|x,w] = exp(wT l x) PK i=0 exp(wT i x) ˆ Y = arg max l P [Y = l|x,w] .
Note that the probabilities sum to 1. Use log-loss and optimize with batch gradient descent. The (negative) likelihood function on an N sampling training set is:
L(w) = − 1 NXNi=1 log PhY = y(i)|x(i),wi
where the sum is over the N points in our training set. Submit answers to the following.
i. Compute (by-hand) the derivative of the log-likelihood of the soft-max function. Write the derivative in terms of conditional probabilities, the vector x, and indicator functions (i.e., do not write this expression in terms of exponentials). You need this gradient in subsequent parts of this problem.
ii. Implement batch gradient descent. What learning rate did you use?
iii. Plot log-loss (i.e., learning curve) of the training set and test set on the same figure. On a separate figure plot the accuracy against iteration number of your model on the training set and test set. Plot each as a function of the iteration number.
iv. Compute the final loss and final accuracy for both your training set and test set.

i)

3 b)
(i)

Log Likelihood

$$L(w) = -\frac{1}{N} \sum_{i=1}^{N} \log P\left[Y = y(i) \mid x(i), w\right]$$

Softmax function - $\frac{\exp(w_c^T x)}{\sum_{i=0}^{K} \exp(w_i^T x)}$

Let $a = w_c^T x$

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial w}$$

$$\boxed{\frac{\partial a}{\partial w} = x}$$

$$\frac{\partial L}{\partial a} = -\frac{1}{N} \sum_{i=1}^{N} \frac{\partial}{\partial a_i}\left[\log\left[\frac{e^{a_j}}{\sum_{j=1}^{K} e^{a_j}}\right]\right]$$

$$= -\frac{1}{N} \sum_{i=1}^{N} \left[\frac{\partial a_j}{\partial a_i} - \frac{\partial}{\partial a_i} \log \sum_{j=1}^{K} e^{a_j}\right]$$

$i = j \Rightarrow$

$$= -\frac{1}{N} \sum_{i=1}^{N} \left[1 - \frac{1}{\sum_{j=1}^{K} e^{a_j}} \sum_{j=1}^{K} \frac{\partial e^{a_j}}{\partial a_i}\right]$$

$$= -\frac{1}{N} \sum_{i=1}^{N} \left[1 - \frac{1}{\sum_{j=1}^{K} e^{a_j}} \left[\sum_{j=1}^{K} e^{a_j} \frac{\partial a_j}{\partial a_i}\right]\right]$$

$$\frac{\partial L}{\partial w} = -\frac{x}{N} \sum_{i=1}^{N} \left[1 - \frac{1}{\sum_{j=1}^{K} e^{a_j}} \left[\sum_{j=1}^{K} e^{a_j} \frac{\partial a_j}{\partial a_i}\right]\right]$$

$i \neq j$

$$= -\frac{1}{N} \sum_{i=1}^{N} \left[0 - \frac{\partial}{\partial a_i} \log\left[\frac{e^{a_j}}{\sum_{j=1}^{K} e^{a_j}}\right]\right]$$

$$= -\frac{1}{N} \sum_{i=1}^{N} \left[\frac{\partial a_j}{\partial a_i} - \frac{\partial}{\partial a_i} \log\left[\sum_{j=1}^{K} e^{a_j}\right]\right]$$

$$= \frac{1}{N} \sum_{i=1}^{N} \left[\frac{e^{a_j}}{\sum_{j=1}^{K} e^{a_j}}\right]$$

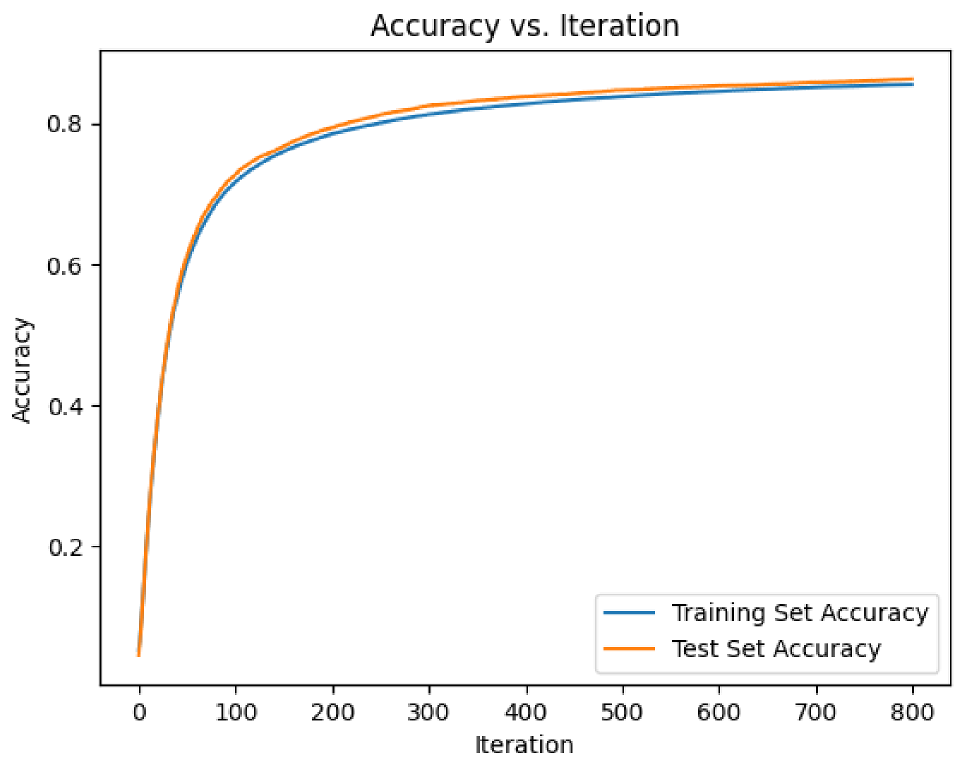
$$\frac{\partial L}{\partial w} = \frac{x}{N} \sum_{i=1}^{N} \left[\frac{e^{a_j}}{\sum_{j=1}^{K} e^{a_j}}\right]$$

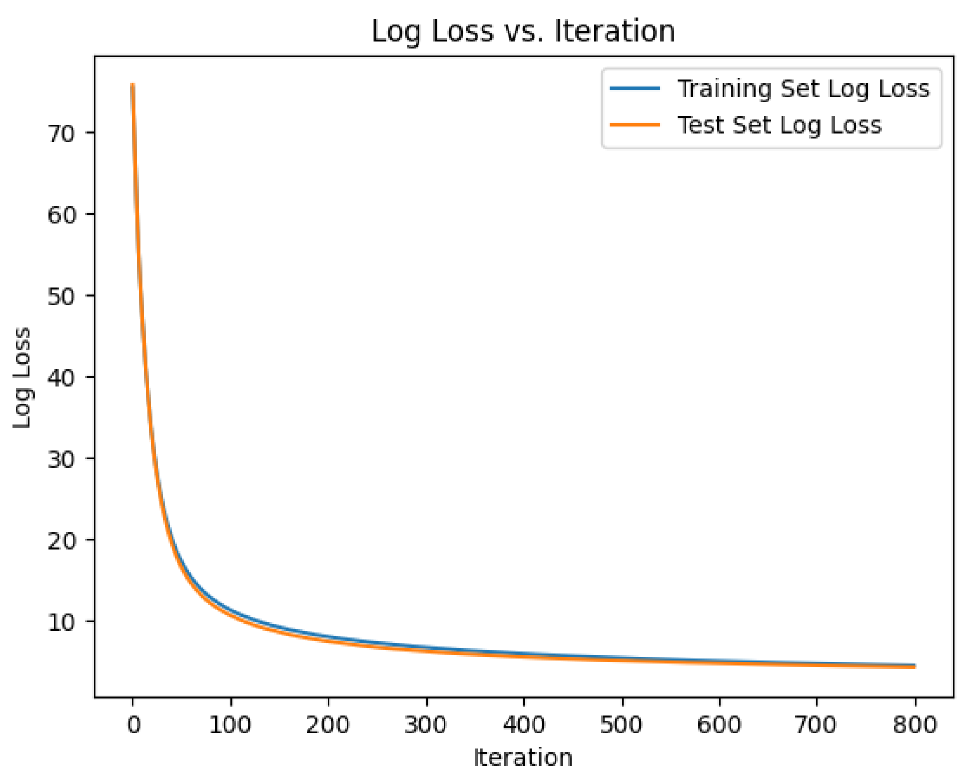
ii)

The outer loop iterates over epochs, whereas the inner loop iterates over training data mini-batches. The gradients are computed and the model parameters (weights w and bias b) are updated in each mini-batch. This is a batch gradient descent procedure in which the gradients are computed and parameter changes are done based on the whole training dataset, split into mini-batches.

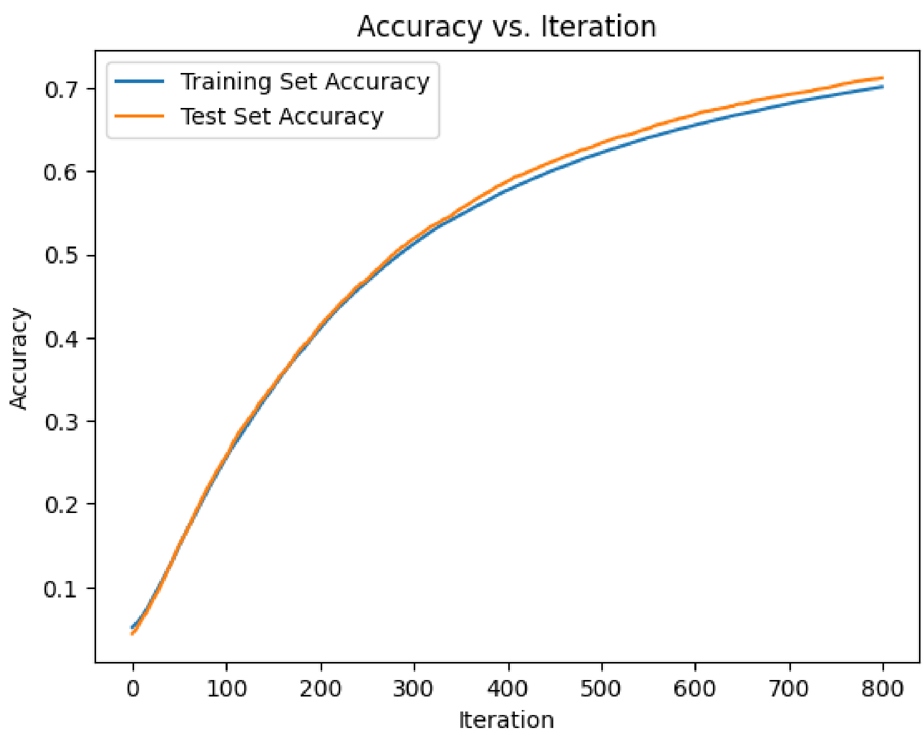
Similar to the 3a there are various learning rates and various epochs to optimize the models, but trial and error method is used to find the better result.

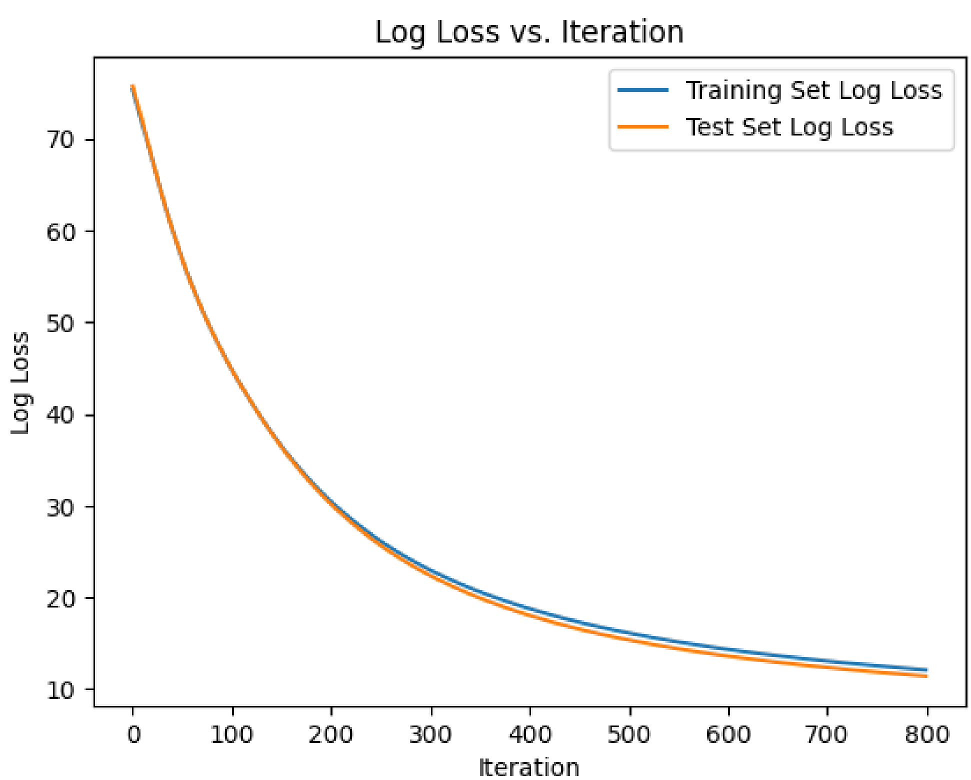
Learning Rate: 0.009
Epochs: 200
Mini Batch: 30


Accuracy vs. Iteration
Accuracy
Iteration
0.2
0.4
0.6
0.8
0
100
200
300
400
500
600
700
800
Training Set Accuracy
Test Set Accuracy

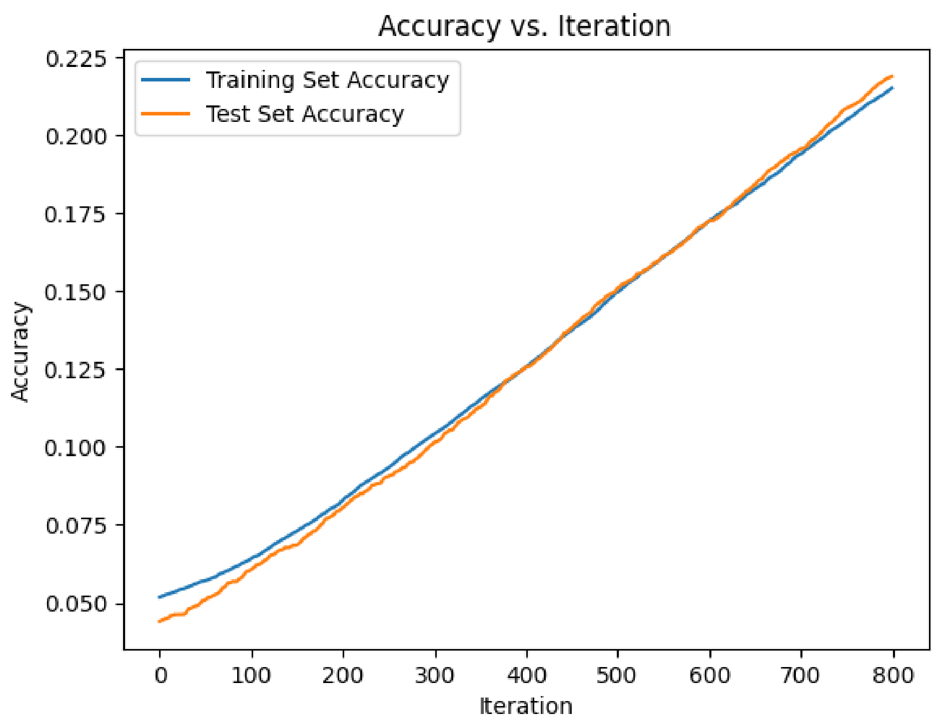

Log Loss vs. Iteration
Log Loss
Iteration
10
20
30
40
50
60
70
0
100
200
300
400
500
600
700
800
Training Set Log Loss
Test Set Log Loss

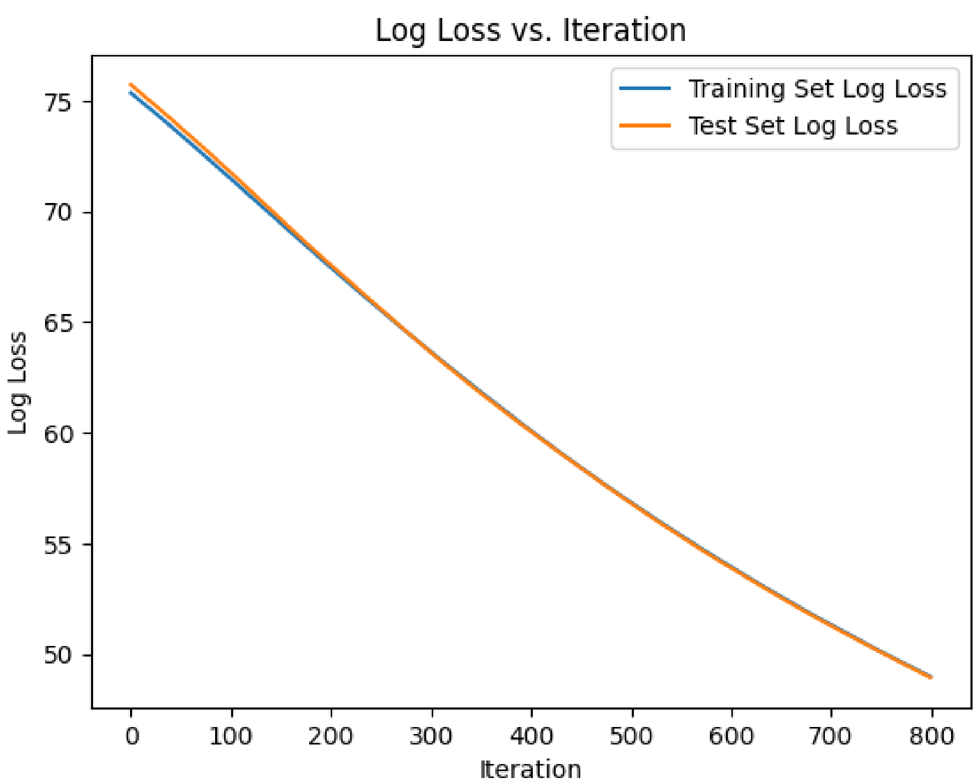
Learning Rate: 0.001
Epochs: 200
Mini Batch: 30





Learning Rate: 0.0001
Epochs: 200
Mini Batch: 30

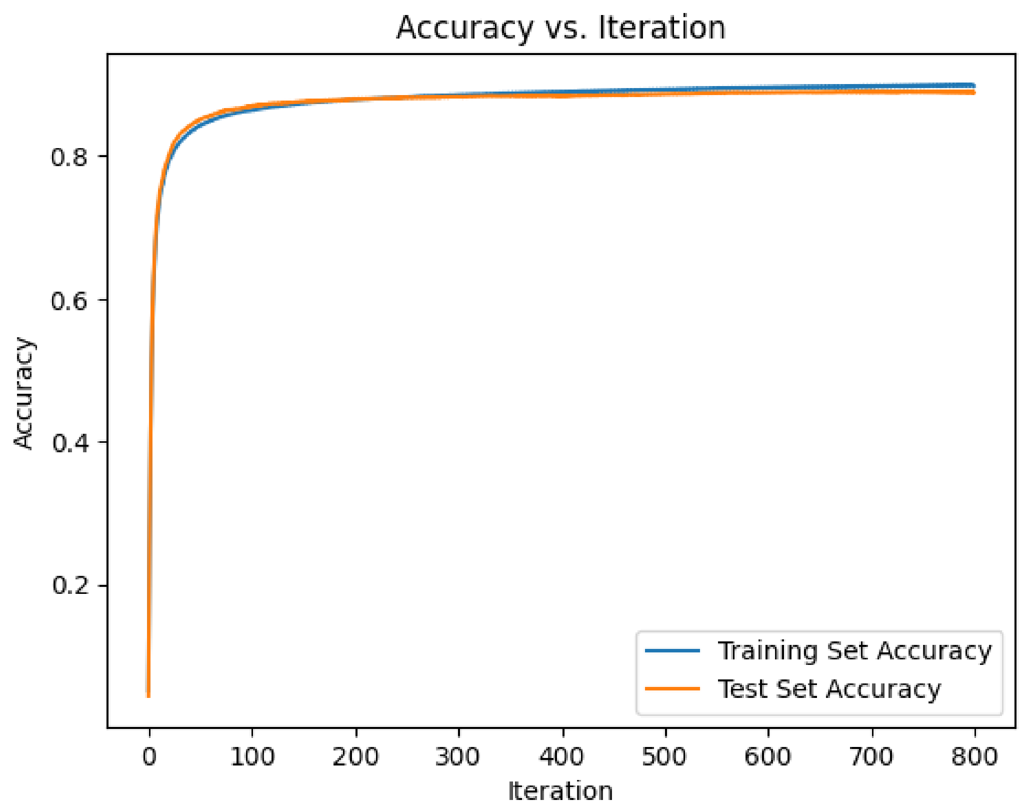


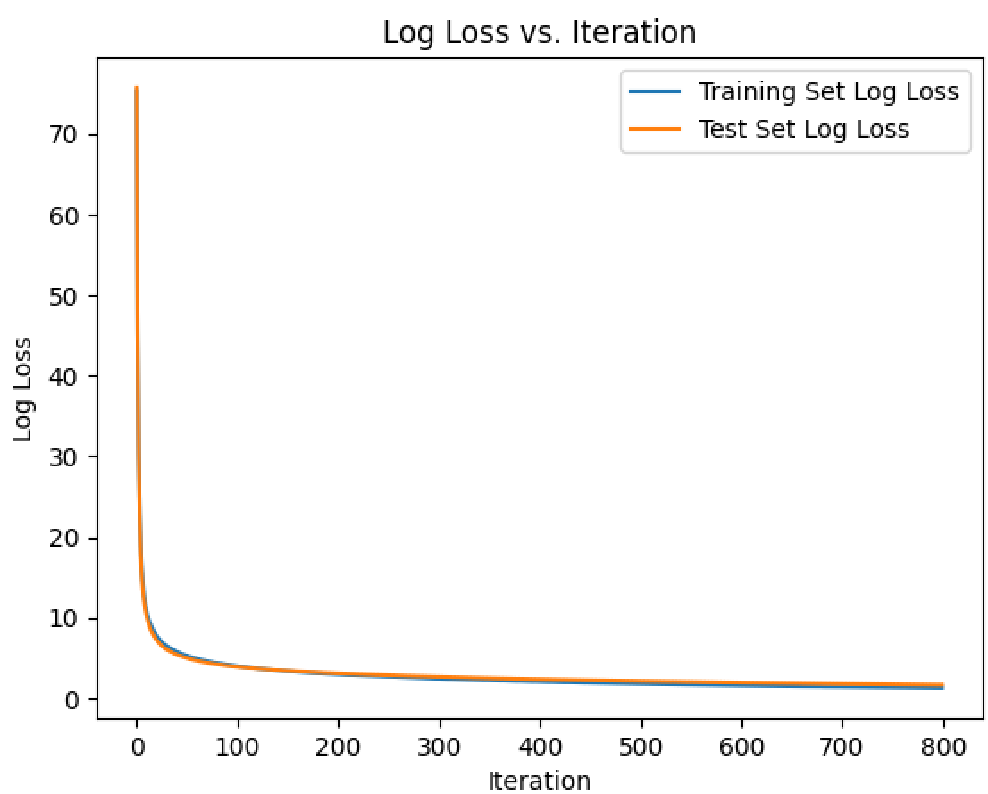


iii)

Learning rate of 0.1 gives the optimized result for the SGD Function.

Learning Rate: 0.1
Epochs: 200
Mini Batch: 30


Accuracy vs. Iteration
Accuracy
Iteration
Training Set Accuracy
Test Set Accuracy
0.2
0.4
0.6
0.8
0
100
200
300
400
500
600
700
800


Log Loss vs. Iteration
Log Loss
Iteration
Training Set Log Loss
Test Set Log Loss
0
10
20
30
40
50
60
70
0
100
200
300
400
500
600
700
800

```
iv)

Final Training Loss: 1.3182062219151651
Final Training Accuracy: 0.9005
Final Test Loss: 1.6650701144677547
Final Test Accuracy: 0.8899
```

APPENDIX:

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

class LogisticRegressionClassifier:
    def __init__(self, learning_rate=0.1, mini_batch=30, n_epochs=200):
        # Initialize the logistic regression classifier with
hyperparameters
        self.learning_rate = learning_rate
        self.mini_batch = mini_batch
        self.n_epochs = n_epochs
        self.losses_train = []  # List to store training losses
        self.losses_test = []   # List to store test losses
        self.accuracies_train = []  # List to store training accuracies
        self.accuracies_test = []   # List to store test accuracies

    def softmax(self, x):
        # Compute the softmax function for input 'x'
        exp_x = np.exp(x - np.max(x, axis=1).reshape(-1, 1))
        return exp_x / np.sum(exp_x, axis=1).reshape(-1, 1)

    def calculate_loss_accuracy(self, x, y, w, b):
        # Calculate loss and accuracy for given data and model parameters
        y_pred = self.softmax(np.dot(x, w) + b.T)
        loss = -np.mean(np.log(np.sum(y_pred * y, axis=1) + 1e-39))
        accuracy = np.mean(np.argmax(y_pred, axis=1) == np.argmax(y,
axis=1))
        return loss, accuracy

    def fit(self, x_train, y_train, x_test, y_test):
        # Fit the logistic regression model to the training data
```

```
        np.random.seed(seed=0)
        w, b = np.random.normal(0, 10, size=(784, 10)),
np.random.randn(10, 1)

        n_iters = int(x_train.shape[0] / self.mini_batch)
        n_updates = 0

        for epoch in range(self.n_epochs):
            for j in range(n_iters):
                if n_updates % 5000 == 0:
                    # Calculate and store losses and accuracies at regular
intervals
                    loss_train, acc_train =
self.calculate_loss_accuracy(x_train, y_train, w, b)
                    loss_test, acc_test =
self.calculate_loss_accuracy(x_test, y_test, w, b)
                    self.losses_train.append(loss_train)
                    self.losses_test.append(loss_test)
                    self.accuracies_train.append(acc_train)
                    self.accuracies_test.append(acc_test)

                n_updates += self.mini_batch
                x_train_batch = x_train[j * self.mini_batch: (j + 1) *
self.mini_batch]
                y_train_batch = y_train[j * self.mini_batch: (j + 1) *
self.mini_batch]

                y_train_pred_batch = self.softmax(np.dot(x_train_batch, w)
+ b.T)
                dw = (1 / self.mini_batch) * np.matmul(x_train_batch.T,
y_train_batch - y_train_pred_batch)
                db = (1 / self.mini_batch) * np.sum(y_train_batch -
y_train_pred_batch, axis=0).reshape(-1, 1)

                w = w + self.learning_rate * dw
                b = b + self.learning_rate * db

        # Calculate final loss and accuracy
        self.final_loss_train, self.final_acc_train =
self.calculate_loss_accuracy(x_train, y_train, w, b)
```

```
        self.final_loss_test, self.final_acc_test =
self.calculate_loss_accuracy(x_test, y_test, w, b)

    def plot_learning_curves(self):
        # Plot learning curves (loss and accuracy)
        plt.plot(self.losses_train, label='Training Loss')
        plt.plot(self.losses_test, label='Test Loss')
        plt.xlabel('Iteration')
        plt.ylabel('Loss')
        plt.title('Loss vs. Iteration')
        plt.legend()
        plt.show()

        plt.plot(self.accuracies_train, label='Training Accuracy')
        plt.plot(self.accuracies_test, label='Test Accuracy')
        plt.xlabel('Iteration')
        plt.ylabel('Accuracy')
        plt.title('Accuracy vs. Iteration')
        plt.legend()
        plt.show()

    def print_final_results(self):
        # Print the final results (loss and accuracy)
        print("Final Training Loss:", self.final_loss_train)
        print("Final Training Accuracy:", self.final_acc_train)
        print("Final Test Loss:", self.final_loss_test)
        print("Final Test Accuracy:", self.final_acc_test)

if __name__ == "__main__":
    # Load training and test data
    data1 = h5py.File('mnist_traindata.hdf5', 'r')
    x_train, y_train = np.asarray(data1['xdata']),
np.asarray(data1['ydata'])

    data2 = h5py.File('mnist_testdata.hdf5')
    x_test, y_test = np.asarray(data2['xdata']),
np.asarray(data2['ydata'])

    # Create binary labels for classifying digit 2
```

```
    y_train_binary = (y_train == [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0]).astype(int)
    y_test_binary = (y_test == [0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0]).astype(int)

    # Create and train the logistic regression model
    model = LogisticRegressionClassifier()
    model.fit(x_train, y_train_binary, x_test, y_test_binary)

    # Plot learning curves and print final results
    model.plot_learning_curves()
    model.print_final_results()
```

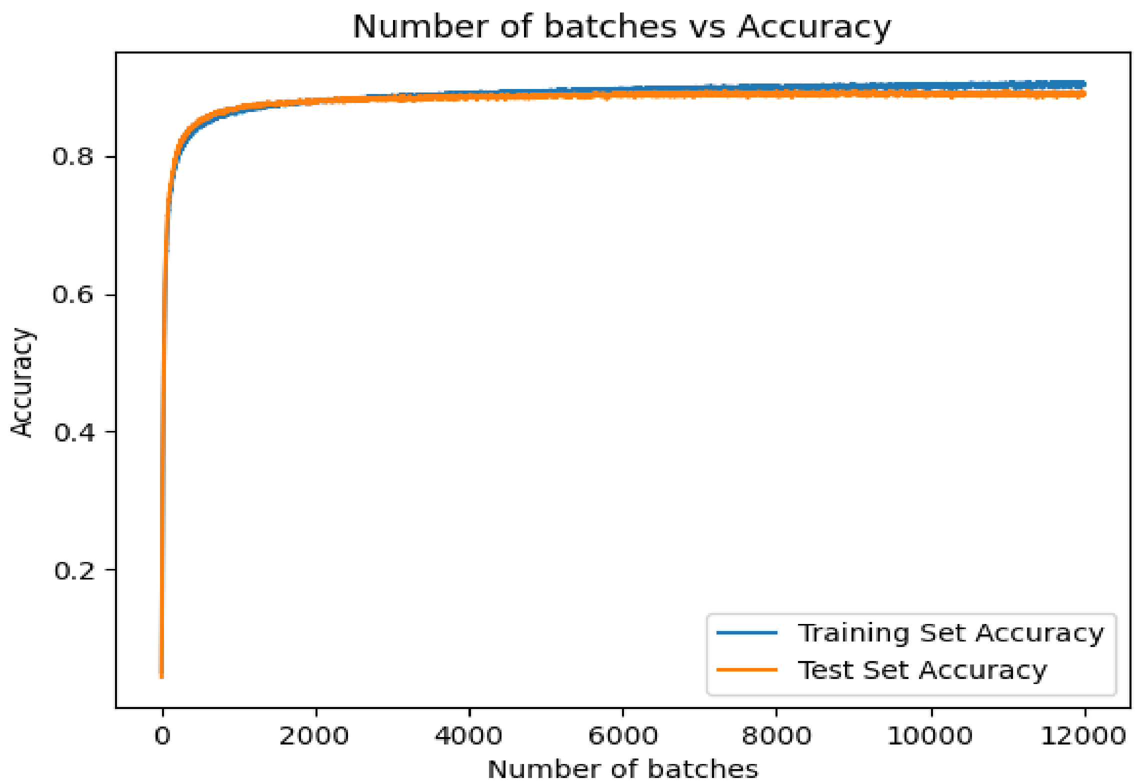
(c) Softmax classification: stochastic gradient descent
In this part you will use stochastic gradient descent (SGD) in place of (deterministic) gradient descent above. Test your SGD implementation using single-point updates and a mini-batch size of 100. You may need to adjust the learning rate to improve performance. You can either: modify the rate by hand or according to some decay scheme or you may choose a single learning rate. You should get a final predictor comparable to that in the previous question. Submit answers to the following.
i. Implement SGD with mini-batch size of 1 (i.e., compute the gradient and update weights after each sample). Record the log-loss and accuracy of the training set and test set every 5,000 samples. Plot the sampled log-loss and accuracy values on the same (respective) figures against the batch number. Your plots should start at iteration 0 (i.e., include initial log-loss and accuracy). Your curves should show performance comparable to batch gradient descent. How many iterations did it take to achieve comparable performance with batch gradient descent? How does this number depend on the learning rate? (or learning rate decay schedule if you have a non-constant learning rate).

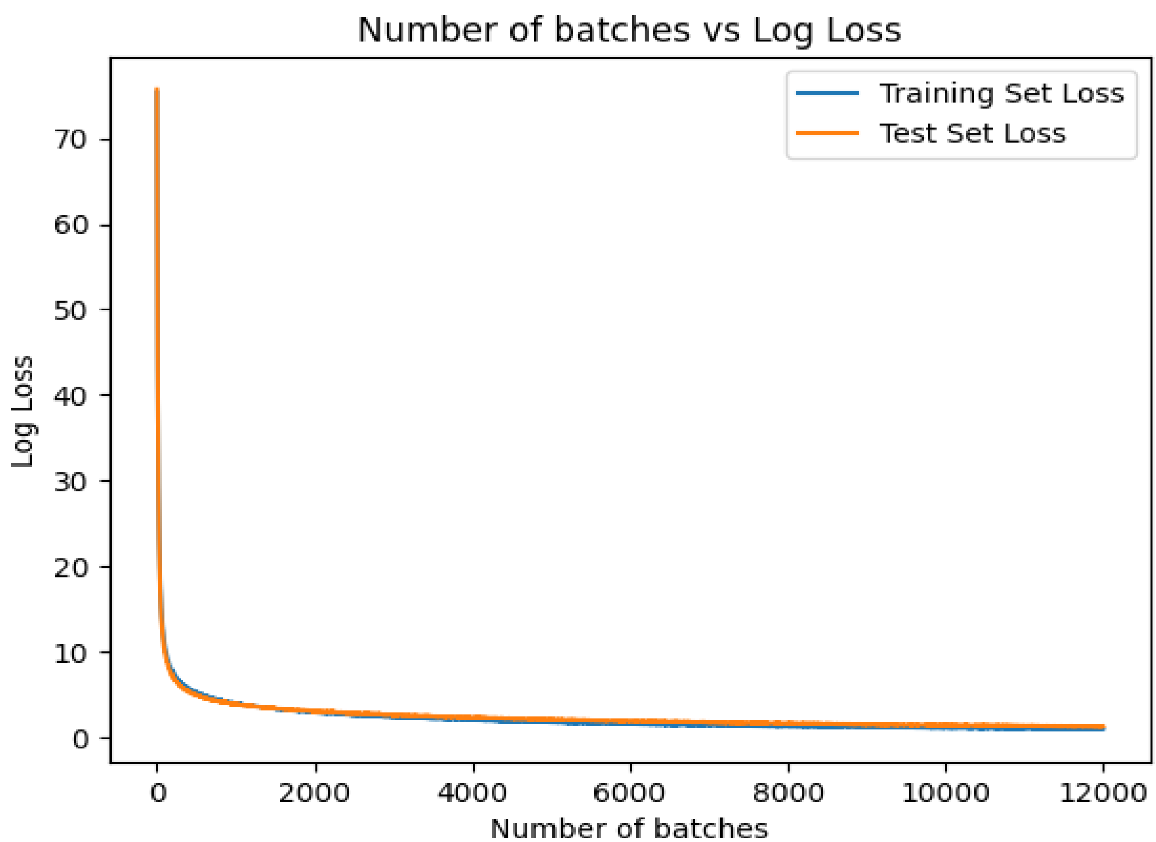
ii. Compare (to batch gradient descent) the total computational complexity to reach a comparable accuracy on your training set. Note that each iteration of batch gradient descent costs an extra factor of N operations where N is the number data points.

iii. Implement SGD with mini-batch size of 100 (i.e., compute the gradient and update weights with accumulated average after every 100 samples). Record the log-loss and accuracies as above (every 5,000 samples – not 5,000 batches) and create similar plots. Your curves should show performance comparable to batch gradient descent. How many iterations did it take to achieve comparable performance with batch gradient descent? How does this number depend on the learning rate? (or learning rate decay schedule if you have a non-constant learning rate).

iv. Compare the computational complexity to reach comparable performance between the 100 sample mini-batch algorithm, the single-point mini-batch, and batch gradient descent. Submit your trained weights to Autolab. Save your weights and bias to an hdf5 file. Use keys W and b for the weights and bias, respectively. W should be a 10×784 numpy array and b should be 10×1 – shape: (10,) – numpy array. The code to save the weights is the same as (a) –substituting W for w.
Note: you will not be scored on your models overall accuracy. But a low-score may indicate errors in training or poor optimization

i)


Number of batches vs Accuracy
Accuracy
0.8
0.6
0.4
0.2
0
2000
4000
6000
8000
10000
12000
Number of batches
Training Set Accuracy
Test Set Accuracy


Number of batches vs Log Loss
Log Loss
70
60
50
40
30
20
10
0
0
2000
4000
6000
8000
10000
12000
Number of batches
Training Set Loss
Test Set Loss

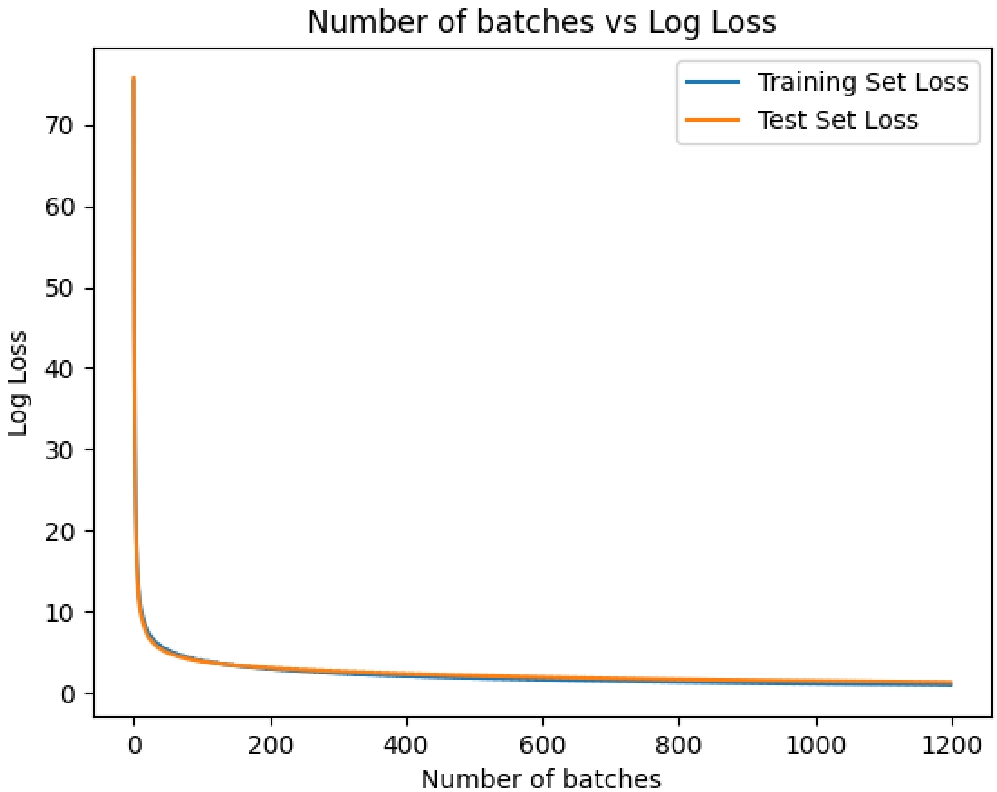
```
For Single Batch:

Final Training Loss: 0.9783524439731436
Final Training Accuracy: 0.9034833333333333
Final Test Loss: 1.3199247740682625
Final Test Accuracy: 0.8918

For BGD:

Final Training Loss: 1.3182062219151651
Final Training Accuracy: 0.9005
Final Test Loss: 1.6650701144677547
Final Test Accuracy: 0.8899
```

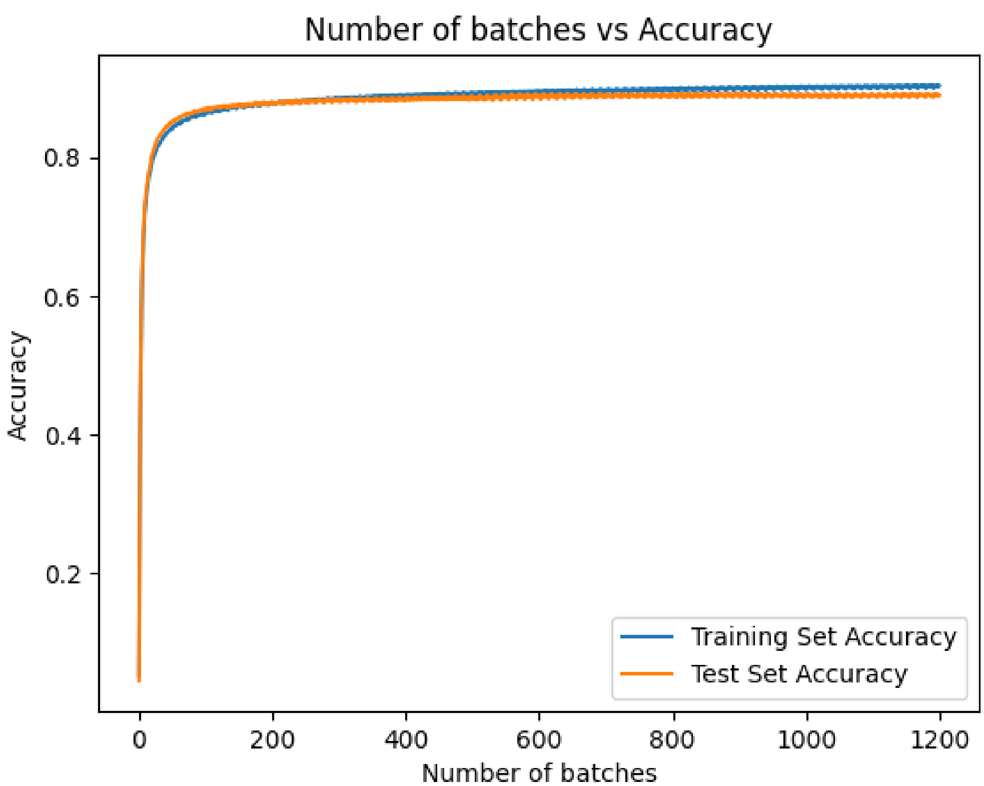
Total Iterations = Number of Epochs * (Number of Training Samples / Mini-Batch Size) = 100 * (1200 / 1) = 120000

A higher learning rate (e.g., 0.1) might result in faster convergence but, if not correctly controlled, can result in overshooting the minimum or divergence.
A lower learning rate (e.g., 0.001) frequently guarantees consistent convergence but may need more iterations to find a satisfactory solution.
Typically, you would experiment with different learning rates and observe performance metrics (e.g., loss, accuracy) on a validation set to identify the number of iterations required for equivalent performance. The ideal learning rate will be determined by the nature of your challenge and dataset.

ii)
In batch gradient descent, the full training dataset is utilized to compute the gradient in each iteration. The number of iterations needed for convergence is determined on the size of the dataset. In single-point mini-batch SGD, just one data point is utilized to compute the gradient in each iteration. The number of iterations necessary for convergence is often substantially more than BGD and is determined by the dataset.

iii)


Number of batches vs Log Loss
Training Set Loss
Test Set Loss
Log Loss
Number of batches
0
10
20
30
40
50
60
70
200
400
600
800
1000
1200


Number of batches vs Accuracy
Training Set Accuracy
Test Set Accuracy
Accuracy
Number of batches
0.2
0.4
0.6
0.8
0
200
400
600
800
1000
1200

```
For 100 mini Batch :

Final Training Loss: 0.9923292968067992
Final Training Accuracy: 0.9045
Final Test Loss: 1.3476918196854346
Final Test Accuracy: 0.8896

For BGD:

Final Training Loss: 1.3182062219151651
Final Training Accuracy: 0.9005
Final Test Loss: 1.6650701144677547
Final Test Accuracy: 0.8899
```

Total Iterations = Number of Epochs * (Number of Training Samples / Mini-Batch Size) = 100 * (1200 / 100) = 1200

iv)

Each iteration of 100-sample mini-batch SGD uses a mini-batch of 100 data points.
The number of iterations required for convergence is often less than that of single-point mini-batch SGD but greater than that of BGD.The main takeaway is that 100-sample mini-batch SGD has a computational complexity similar to batch gradient descent (BGD), but it converges faster because it takes advantage of some of the advantages of both BGD (stable updates) and single-point mini-batch SGD (frequent updates for better convergence).The number of iterations necessary for convergence will continue to be determined by variables such as the dataset, network design, and learning rate.

APPENDIX:

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

class MiniBatchSGD:
    def __init__(self, learning_rate=0.01, mini_batch_size=1,
n_epochs=100):
        # Initialize hyperparameters and model variables
        self.learning_rate = learning_rate
```

```
        self.mini_batch_size = mini_batch_size
        self.n_epochs = n_epochs
        self.loss_train_list = []
        self.loss_test_list = []
        self.acc_train_list = []
        self.acc_test_list = []

        # Load the training and test datasets
        data1 = h5py.File('mnist_traindata.hdf5', 'r')
        data2 = h5py.File('mnist_testdata.hdf5')
        self.xtrain = np.asarray(data1['xdata'])
        self.ytrain = np.asarray(data1['ydata'])
        self.xtest = np.asarray(data2['xdata'])
        self.ytest = np.asarray(data2['ydata'])

        # Initialize weights and biases
        np.random.seed(seed=0)
        self.w = np.random.normal(0, 10, size=(784, 10))
        self.b = np.random.randn(10, 1)

    def softmax(self, x):
        # Softmax function to compute probabilities
        a = np.exp(x - np.max(x, axis=1).reshape(-1, 1))
        return a / np.sum(a, axis=1).reshape(-1, 1)

    def train(self):
        n_samples = self.xtrain.shape[0]

        for epoch in range(self.n_epochs):
            # Shuffle the data at the beginning of each epoch (optional)
            permutation = np.random.permutation(n_samples)
            xtrain_shuffled = self.xtrain[permutation]
            ytrain_shuffled = self.ytrain[permutation]

            for j in range(0, n_samples, self.mini_batch_size):
                x_train_batch = xtrain_shuffled[j:j +
self.mini_batch_size]
                y_train_batch = ytrain_shuffled[j:j +
self.mini_batch_size]
```

```
                # Perform weight update for the mini-batch
                y_train_pred_batch = self.softmax(np.dot(x_train_batch,
self.w) + self.b.T)
                dw = np.outer(x_train_batch.T, (y_train_batch -
y_train_pred_batch).T)
                db = np.sum((y_train_batch - y_train_pred_batch).T,
axis=1).reshape(-1, 1)

                self.w = self.w + self.learning_rate * dw
                self.b = self.b + self.learning_rate * db

                if j % 500 == 0:
                    # Calculate and store loss and accuracy at selected
intervals
                    y_train_pred = self.softmax(np.dot(self.xtrain,
self.w) + self.b.T)
                    y_test_pred = self.softmax(np.dot(self.xtest, self.w)
+ self.b.T)
                    loss_train = -np.mean(np.log(np.sum(y_train_pred *
self.ytrain, axis=1) + 1e-39))
                    loss_test = -np.mean(np.log(np.sum(y_test_pred *
self.ytest, axis=1) + 1e-39))
                    acc_train = np.mean(np.argmax(y_train_pred, axis=1) ==
np.argmax(self.ytrain, axis=1))
                    acc_test = np.mean(np.argmax(y_test_pred, axis=1) ==
np.argmax(self.ytest, axis=1))
                    self.loss_train_list.append(loss_train)
                    self.loss_test_list.append(loss_test)
                    self.acc_train_list.append(acc_train)
                    self.acc_test_list.append(acc_test)

            print("Epoch:", epoch)
            print("Train Loss:", loss_train)
            print("Test Loss:", loss_test)

    def evaluate(self):
        # Compute final loss and accuracy for training set
        y_train_pred = self.softmax(np.dot(self.xtrain, self.w) +
self.b.T)
```

```
        final_loss_train = -np.mean(np.log(np.sum(y_train_pred *
self.ytrain, axis=1) + 1e-39))
        final_acc_train = np.mean(np.argmax(y_train_pred, axis=1) ==
np.argmax(self.ytrain, axis=1))

        # Compute final loss and accuracy for test set
        y_test_pred = self.softmax(np.dot(self.xtest, self.w) + self.b.T)
        final_loss_test = -np.mean(np.log(np.sum(y_test_pred * self.ytest,
axis=1) + 1e-39))
        final_acc_test = np.mean(np.argmax(y_test_pred, axis=1) ==
np.argmax(self.ytest, axis=1))

        print("Final Training Loss:", final_loss_train)
        print("Final Training Accuracy:", final_acc_train)
        print("Final Test Loss:", final_loss_test)
        print("Final Test Accuracy:", final_acc_test)

    def plot_loss_accuracy(self):
        # Plot training and test loss
        plt.plot(self.loss_train_list, label='Training Set Loss')
        plt.plot(self.loss_test_list, label='Test Set Loss')
        plt.xlabel('Number of batches')
        plt.ylabel('Log Loss')
        plt.title('Number of batches vs Log Loss')
        plt.legend()
        plt.show()

        # Plot training and test accuracy
        plt.plot(self.acc_train_list, label='Training Set Accuracy')
        plt.plot(self.acc_test_list, label='Test Set Accuracy')
        plt.xlabel('Number of batches')
        plt.ylabel('Accuracy')
        plt.title('Number of batches vs Accuracy')
        plt.legend()
        plt.show()

if __name__ == "__main__":
    model = MiniBatchSGD(learning_rate=0.01, mini_batch_size=1,
n_epochs=100)
    model.train()
```

```
model.evaluate()
model.plot_loss_accuracy()
```