Student Name: Mariem Ahmed

ECE421 - Winter 2021

Assignment 1: Logistic Regression

Due date: February 9, 2021

Submissions: Submit both your report (a single PDF file) and all codes on Quercus.

Objectives: In this assignment, you will first implement a simple logistic regression classifier using Numpy and train your model by applying (Stochastic) Gradient Descent algorithm. Next, you will implement the same model, this time in TensorFlow and use Stochastic Gradient Descent and ADAM to train your model. You are encouraged to look up TensorFlow APIs for useful utility functions, at: https://www.tensorflow.org/api_docs/python/ (https://www.tensorflow.org/api_docs/python/).

In [ ]:
```
import tensorflow as tf
import tensorflow.compat.v1 as tf
import numpy as np
import matplotlib.pyplot as plt
import time

def loadData():
    with np.load('notMNIST.npz') as data :
        Data, Target = data ['images'], data['labels']
        posClass = 2
        negClass = 9
        dataIndx = (Target==posClass) + (Target==negClass)
        Data = Data[dataIndx]/255.
        Target = Target[dataIndx].reshape(-1, 1)
        Target[Target==posClass] = 1
        Target[Target==negClass] = 0
        np.random.seed(521)
        randIndx = np.arange(len(Data))
        np.random.shuffle(randIndx)
        Data, Target = Data[randIndx], Target[randIndx]
        trainData, trainTarget = Data[:3500], Target[:3500]
        validData, validTarget = Data[3500:3600], Target[3500:3600]
        testData, testTarget = Data[3600:], Target[3600:]
    return trainData, validData, testData, trainTarget, validTarget, tes
tTarget
```

1 Logistic Regression with Numpy[20 points]

Logistic regression is one the most widely used linear classification models in machine learning. In logistic regression, we model the probability of a sample $\mathbf{x}$ belonging to the positive class as

$$\hat{y}(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b),$$

where $z = \mathbf{w}^\top \mathbf{x} + b$, also called *logit*, is basically the linear transformation of input vector $\mathbf{x}$ using weight vector $\mathbf{w}$ and bias scalar b, and $\sigma(z) = 1/(1 + \exp(-z))$ is the sigmoid or logistic function. The sigmoid function "squashes" the real-valued logits to fall between zero and one.

The cross-entropy loss $\mathcal{L}_{\text{CE}}$ and the regularization term $\mathcal{L}_{\text{w}}$ will form the total loss function as:

$$\begin{aligned}\mathcal{L} =& \mathcal{L}_{\text{CE}} + \mathcal{L}_{\text{w}} \\ =& \frac{1}{N} \sum_{n=1}^{N} \left[- y^{(n)} \log \hat{y}\left(\mathbf{x}^{(n)}\right) - \left(1 - y^{(n)}\right) \log \left(1 - \hat{y}\left(\mathbf{x}^{(n)}\right)\right) \right] + \frac{\lambda}{2} \|\mathbf{w}\|_2^2\end{aligned}$$

Note that $y^{(n)} \in \{0, 1\}$ is the class label for the n-th training image and λ is the regularization parameter.

1. Loss Function and Gradient [8 pts]:

Implement two vectorized Numpy functions (i.e. avoid using for loops by employing matrix products and broadcasting) to compute the loss function and its gradient. The grad_loss function should compute and return an analytical expression of the gradient of the loss with respect to both the weights and bias. Both function headers are below. Include the analytical expressions in your report as well as a snippet of your Python code.

Let's Find the Gradient with respect to w and b

To determine the gradient of the loss function with respect to W, substitute $\hat{y}(\mathbf{x}) = \frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}$ into the total loss function:

$$\mathcal{L} = \frac{1}{N}\sum_{n=1}^{N}[-y^{(n)} log\ (\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}^{(n)}+b)}}) - (1-y^{(n)})\log\ (1-\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}^{(n)}+b)}})] + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

$$= \frac{1}{N}\sum_{n=1}^{N}[y^{(n)} log\ \left(1+e^{-(\mathbf{w}^T\mathbf{x}^{(n)}+b)}\right) + (1-y^{(n)})\log\ (1+e^{(\mathbf{w}^T\mathbf{x}^{(n)}+b)})] + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

Then, the gradient of the loss function with respect to W can be

$$\nabla_{\mathbf{w}}(\mathcal{L}) = \frac{1}{N}\sum_{n=1}^{N}[-y^{(n)}\frac{\mathbf{x}^{(n)}+b}{1+e^{(\mathbf{w}^T\mathbf{x}^{(n)}+b)}} + (1-y^{(n)})\frac{\mathbf{x}^{(n)}+b}{1+e^{-(\mathbf{w}^T\mathbf{x}^{(n)}+b)}}] + \lambda\mathbf{w}$$

This can be re-written as

$$\nabla_{\mathbf{w}}(\mathcal{L}) = \frac{1}{N}\sum_{n=1}^{N}[(\frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}^{(n)}+b)}} - y^{(n)})\mathbf{x}^{(n)}] + \lambda\mathbf{w}$$

$$= \frac{1}{N}(\mathbf{x}^T(\hat{y}(\mathbf{x}) - y)) + \lambda\mathbf{w}$$

The gradient of the loss function with respect to b can be

$$\nabla_b(\mathcal{L}) = \frac{1}{N}\sum_{n=1}^{N}[\frac{y^{(n)}}{1+e^{(\mathbf{w}^T\mathbf{x}^{(n)}+b)}} - \frac{1-y^{(n)}}{1+e^{-(\mathbf{w}^T\mathbf{x}^{(n)}+b)}}]$$

In [ ]:
```
# The loss fuction should compute and return the Total Loss Function

def loss(w, b, x, y, reg):
    ŷ = 1.0/(1.0+np.exp(-(np.matmul(x,w)+b)))
    L_ce = (np.sum(-(y*np.log(ŷ)+(1-y)*np.log(1-ŷ))))/(np.shape(y)[0]) #
The cross-entropy loss LCE
    L_w = reg/2*np.sum(w*w) #The regularization term Lw
    Total_Loss_Function = L_ce + L_w #The total loss function
    return Total_Loss_Function

# The grad_loss function should compute and return an analytical express
ion of the
# gradient of the loss with respect to both the weights and bias.

def grad_loss(w, b, x, y, reg):
    ŷ = 1.0/(1.0+np.exp(-(np.matmul(x,w)+b)))
    derivative_wrt_Lce = np.matmul(np.transpose(x), (ŷ - y))/(np.shape(y
)[0])
    derivative_wrt_Lw = 2*reg*w
    grad_wrt_w = derivative_wrt_Lce + derivative_wrt_Lw
    grad_wrt_b = np.sum((ŷ - y))/(np.shape(y)[0])
    return grad_wrt_w, grad_wrt_b
```

2. Gradient Descent Implementation [6 pts]:

Using the gradient computed from part 1, implement the batch Gradient Descent algorithm to classify the two classes in the notMNIST dataset. The function should accept 8 arguments - the weight vector, the bias, the data matrix, the labels, the learning rate, the number of epochs, λ and an error tolerance (set to 1 × 10^−7). The training should stop if the total number of epochs is reached, or the difference between the old and updated weights are smaller than the error tolerance. The function should return the optimized weight vector and bias. The function header is below.

In [ ]:
```
# Here is the implementation of the batch Gradient Descent algorithm to
 classify the two classes in the notMNIST dataset.
# The function should accept 8 arguments which are the weight vector, th
e bias, the data matrix, the labels, the learning rate,
# the number of epochs, λ and an error tolerance (set to 1 × 10^-7).
# The training should stop if the total number of epochs is reached, or
 the difference between the old and updated weights are
# smaller than the error tolerance.
# The function should return the optimized weight vector and bias.

def grad_descent(w, b, x, y, alpha, epochs, reg, error_tol):
    for i in range(epochs):
        grad_wrt_w, grad_wrt_b = grad_loss(w, b, x, y, reg)
        new_value_of_w = w - alpha * grad_wrt_w
        new_value_of_b = b - alpha * grad_wrt_b
        normalize = np.linalg.norm(new_vale_of_w)
        if normalize < error_tol:
            return new_value_of_w
        else:
            w = new_value_of_w
            b = new_value_of_b
    return w,b
```

You may also wish to print and/or store the training, validation and test losses/accuracies in this function for plotting. (In this case, you can add two more inputs for validation and test data to grad_descent()).

In [ ]:
```
# Here is testing the implementation of Gradient Descent with 5000 epoch
s and λ = 0. Investigate the
# impact of learning rate, α = {0.005, 0.001, 0.0001} on the performance
of your classifier. Plot
# the training and validation loss (on one figure) vs. number of passed
 epochs for each value of
# α. Repeat this for training and validation accuracy.
# Uisng this webiste to help in getting some information about Logistic
 Regression with Numpy (np.matmul, ___.append)
# [https://www.kaggle.com/emilyhorsman/basic-logistic-regression-with-nu
mpy], [https://github.com/m4qo5/logistic-regression-numpy]

# Later we will investigate the impact of regularization by modifying th
e regularization parameter, λ = {0.001, 0.1, 0.5}
# for α = 0.005. Plot the training/validation loss/accuracy vs. epochs f
igure, similar to the previous part.

def grad_descent(w, b, x, y, alpha, epochs, reg, error_tol, validData, t
estData, validTarget, testTarget):
        epochs = 700;
        validation_loss = [loss(w, b, validData, validTarget, reg)]
        testing_loss = [loss(w, b, testData, testTarget, reg)]
        training_loss = [loss(w, b, x, y, reg)]
        # Training Computation
        training = np.matmul(trainData,w)+b
        training_accuracy = [np.sum((training>=0.5)==trainTarget)/(train
Data.shape[0])]
        # Validation Computation
        validation = np.matmul(validData,w)+b
        validation_accuracy = [np.sum((validation>=0.5)==validTarget)/(v
alidTarget.shape[0])]
        # Testing Computation
        testing = np.matmul(testData,w)+b
        testing_accuracy = [np.sum((testing>=0.5)==testTarget)/(testData
.shape[0])]
        # training model
        for i in range(epochs):
            grad_wrt_w, grad_wrt_b = grad_loss(w, b, x, y, reg)
            new_value_of_w = w - alpha * grad_wrt_w
            new_value_of_b = b - alpha * grad_wrt_b
            normalize = np.linalg.norm(new_value_of_w)
            training_loss.append(loss(new_value_of_w,new_value_of_b,x, y
, reg))
            validation_loss.append(loss(new_value_of_w,new_value_of_b,va
lidData,validTarget,reg))
            testing_loss.append(loss(new_value_of_w,new_value_of_b,testD
ata,testTarget,reg))
            # Training Computation for new dataset
            training = np.matmul(trainData,new_value_of_w)+ new_value_of
_b
            training_accuracy.append(np.sum((training>=0.5)==trainTarget
)/(trainData.shape[0]))
            # Validation Computation for new dataset
            validation = np.matmul(validData,new_value_of_w)+new_value_o
f_b
            validation_accuracy.append(np.sum((validation>=0.5)==validTa
```

```
rget)/(validTarget.shape[0]))
            # Testing Computation for new dataset
            testing = np.matmul(testData,new_value_of_w)+new_value_of_b
            testing_accuracy.append(np.sum((testing>=0.5)==testTarget)/(
testData.shape[0]))
            normalize = np.linalg.norm(new_value_of_w-w)
            if normalize < error_tol:
                return new_value_of_w,new_value_of_b, training_loss, val
idation_loss, testing_loss, training_accuracy, validation_accuracy, test
ing_accuracy
            else:
                w = new_value_of_w
                b = new_value_of_b
        return w,b,training_loss, validation_loss, testing_loss, trainin
g_accuracy, validation_accuracy, testing_accuracy

# Used this webiste to come up with the following below
# [https://github.com/geyang/deep_learning_notes/blob/master/Ch3%20Advan
ced%20Tensorflow/2%20-%20MNIST%20Logistic%20Regression%20L2%20Regulariza
tion.py]

# Initializing Parameters
# Here you can change the parameters for reg which is Lamda and alpha wh
ich is the learning rates
μ = 0
σ = 0.5 # mean and standard deviation
b = 0
alpha = 0.0001
epochs = 5000
reg = 0
error_tol = 1e-7

if __name__ == "__main__":
    trainData, validData, testData, trainTarget, validTarget, testTarget
= loadData()
    trainData = trainData.reshape((trainData.shape[0], trainData.shape[1
]*trainData.shape[2]))
    validData = validData.reshape((-1,validData.shape[1]*validData.shape
[2]))
    testData = testData.reshape((-1,testData.shape[1]*testData.shape[2
]))
    w = np.random.normal(μ, σ, (trainData.shape[1],1))
    w, b, training_loss, validation_loss, testing_loss, training_accurac
y, validation_accuracy, testing_accuracy = grad_descent(w, b, trainData,
trainTarget, alpha, epochs, reg, error_tol, validData, testData, validTa
rget, testTarget)
    training = np.matmul(trainData,w)+b
    print("Training Data Accuracy: ", np.sum((training>=0.5)==trainTarge
t)/(trainData.shape[0]))
    validation = np.matmul(validData,w)+b
    print("Validation Data Accuracy: ", np.sum((validation>=0.5)==validT
arget)/(validData.shape[0]))
    testing = np.matmul(testData,w)+b
    print("Testing Data Accuracy: ", np.sum((testing>=0.5)==testTarget)/
(testData.shape[0]))
    epochs = range(len(training_loss))
```

```
    # Using this website for plotting [https://matplotlib.org/tutorials/
introductory/pyplot.html]
    plt.subplot(121)
    plt.plot(epochs,training_loss)
    plt.plot(epochs,validation_loss)
    plt.plot(epochs,testing_loss)
    plt.suptitle('Logistic Regression for α = %s  λ = %s' %(alpha,  reg),
fontsize=16)
    plt.legend(['Training Loss', 'Validation Loss', 'Testing Loss'], loc
='upper right')
    plt.subplot(122)
    plt.plot(epochs,training_accuracy)
    plt.plot(epochs,validation_accuracy)
    plt.plot(epochs,testing_accuracy)
    plt.legend(['Training Accuracy', 'Validation Accuracy', 'Testing Acc
uracy'], loc='lower right')
    plt.show()
```

3. Tuning the Learning Rate [3 pts]:

Test your implementation of Gradient Descent with 5000 epochs and λ = 0. Investigate the impact of learning rate, α = {0.005, 0.001, 0.0001} on the performance of your classifier. Plot the training and validation loss (on one figure) vs. number of passed epochs for each value of α. Repeat this for training and validation accuracy. You should submit a total of 6 figures in your report for this part. Also, explain how you choose the best learning rate, and what accuracy you report for the selected learning rate.

Answer: The weight and bias using Gaussian distribution with mean 0 and and standard deviation 0.5, and $\lambda = 0$. The Figure 1 below shows the learning curves and accuracy using different learning rates among training, validation and testing. Below are the aprroximated final accuracy values for each learning rate. According to the results, the model performs the best when $\alpha = 0.005$ because it has the highest accuracy and lowest loss on both training and validation sets and also converges faster than the other two. I can also see that $\alpha = 0.0001$ has the lowest quality result comparing to the other two because it goes very slow as approaching the global minimum.

1. Logistic Regression for $\alpha = 0.005$ and $\lambda = 0$

- Training Data Accuracy: 0.938
- Validation Data Accuracy: 0.92
- Testing Data Accuracy: 0.9655172413793104

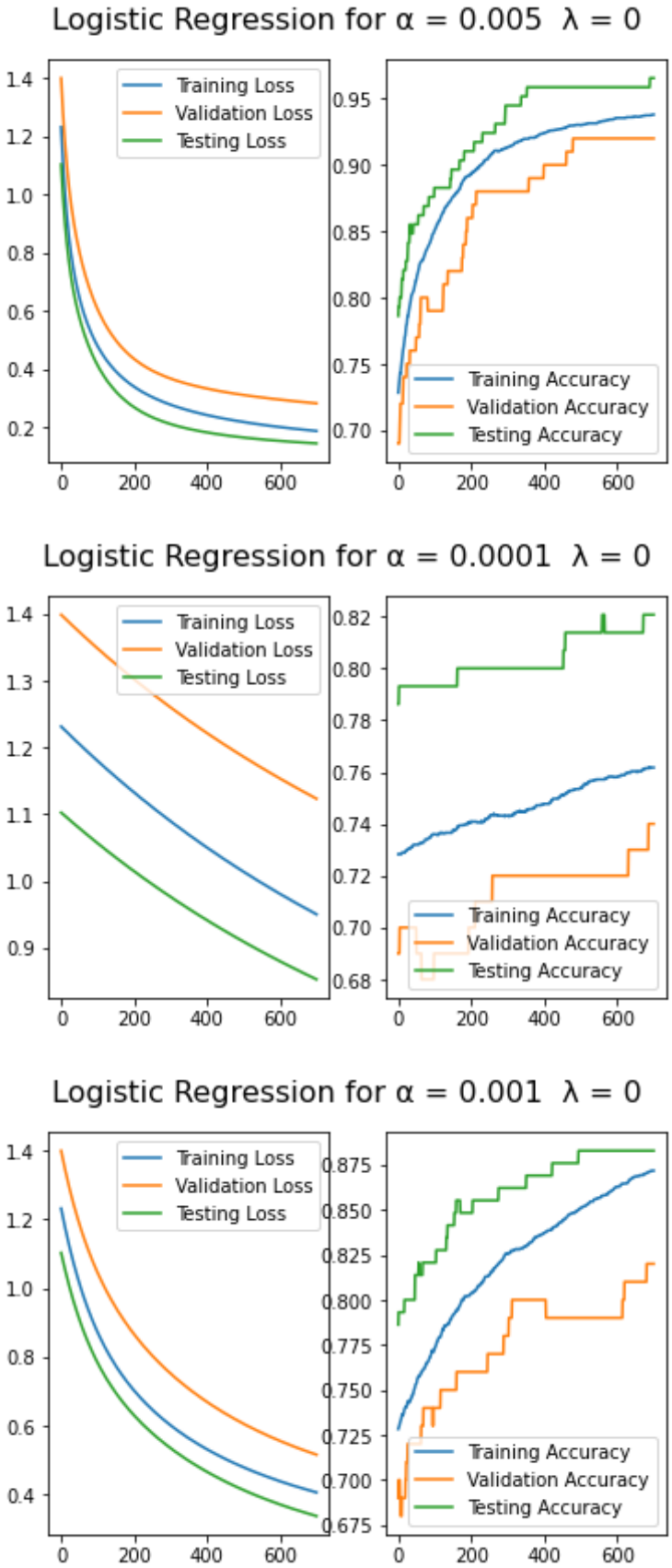
1. Logistic Regression for $\alpha = 0.0001$ and $\lambda = 0$

- Training Data Accuracy: 0.7617142857142857
- Validation Data Accuracy: 0.74
- Testing Data Accuracy: 0.8206896551724138

1. Logistic Regression for $\alpha = 0.001$ and $\lambda = 0$

- Training Data Accuracy: 0.8717142857142857
- Validation Data Accuracy: 0.82
- Testing Data Accuracy: 0.8827586206896552

Figure 1: Training, Validation and Testing loss with respect to different Learning Rate (α) and epochs 5000 and error_tol 1e-7.



4. Generalization [3 pts]:

Investigate the impact of regularization by modifying the regularization parameter, λ = {0.001, 0.1, 0.5} for α = 0.005. Plot the training/validation loss/accuracy vs. epochs figure, similar to the previous part. Also, explain how you choose the best parameter, and what accuracy you report for the selected model.

Answer: Figure 2 below shows the accuracy using different weights for each sets. Below are the approximated values for the final accuracy for training, validation and testing sets. I noticed that when λ = 0.5, it shows the best model because it has the fastest to converge at the end and lowest value validation loss and also in other sets too. But for λ = 0.1, it has lowest validation and testing results for loss and highest values for validation and testing data results for accuracy. I can conclude that small λ values in this model prevents over-fitting and improve its convergence.

1. Logistic Regression for α = 0.005 and λ = 0.5

- Training Data Accuracy: 0.9585714285714285
- Validation Data Accuracy: 0.96
- Testing Data Accuracy: 0.9586206896551724

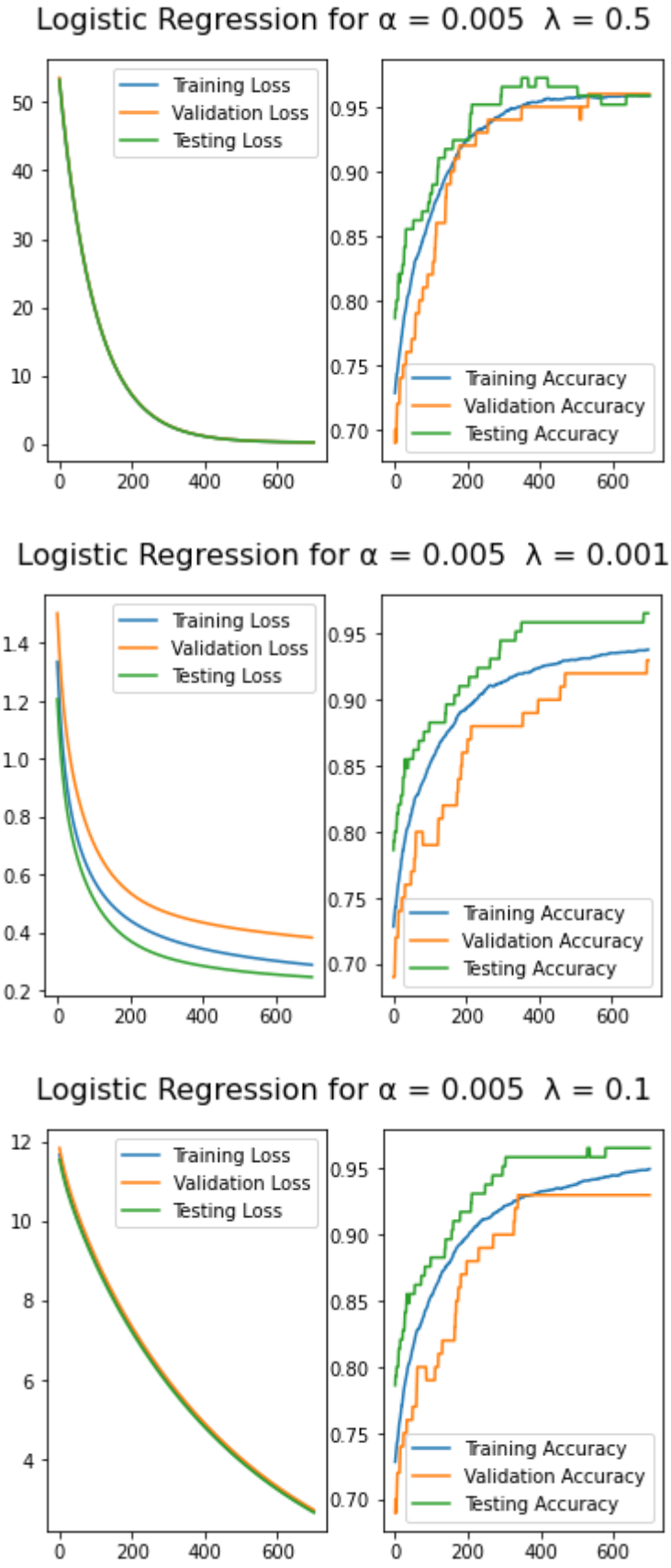
1. Logistic Regression for α = 0.005 and λ = 0.001

- Training Data Accuracy: 0.938
- Validation Data Accuracy: 0.93
- Testing Data Accuracy: 0.9655172413793104

1. Logistic Regression for α = 0.005 and λ = 0.1

- Training Data Accuracy: 0.9497142857142857
- Validation Data Accuracy: 0.93
- Testing Data Accuracy: 0.9655172413793104

Figure 2: Training, Validation and Testing loss with respect to different Weights (λ) and epochs 5000 and errorr_tol 1-e-7.



2 Logistic Regression in TensorFlow [20 points]

In the exercises above, you implemented the Batch Gradient Descent Algorithm. For large datasets however, obtaining the loss gradient using *all* the training data at each iteration may be infeasible. Stochastic Gradient Descent, or Mini-batch gradient descent is aimed at solving this problem. You will be implementing the SGD algorithm and optimizing the training process using the Adaptive Moment Estimation technique (Adam), using TensorFlow.

1. Building the Computational Graph [5 pts]:

Define a function, buildGraph() that initializes the TensorFlow computational graph. To do so, you must initialize the following:

(a) The weight and bias tensors: for the weight tensors, use tf truncated_normal and set the standard deviation to 0.5. Initial the bias variable with zero.

(b) Placeholders for data, labels and λ: use tf.placeholder.

(c) The loss tensor: Calculates the CE loss function with the regularization term. You may wish to investigate the TensorFlow API Documentation regarding losses and regularization on their website.

(d) The optimizer: use tf.train.AdamOptimizer to minimize the total loss. Set the learning rate α to 0.001. The function should return the TensorFlow objects for weight, bias, predicted labels, real labels, the loss, and the optimizer.

In [ ]:
```
import tensorflow as tf
import tensorflow.compat.v1 as tf
import numpy as np
import matplotlib.pyplot as plt
import time

# The notMNIST dataset is a image recognition dataset of font glyphs for the letters A through J
# useful with simple neural networks. It is quite similar to the classic MNIST dataset of handwritten
# digits 0 through 9. We use the following script to generate a smaller dataset that only contains the
# images from two letter classes: “C”(the positive class) and “J”(the negative class). This smaller
# subset of the data contains 3500 training images, 100 validation images and 145 test images.

def loadData():
    with np.load('notMNIST.npz') as data :
        Data, Target = data ['images'], data['labels']
        posClass = 2
        negClass = 9
        dataIndx = (Target==posClass) + (Target==negClass)
        Data = Data[dataIndx]/255.
        Target = Target[dataIndx].reshape(-1, 1)
        Target[Target==posClass] = 1
        Target[Target==negClass] = 0
        np.random.seed(521)
        randIndx = np.arange(len(Data))
        np.random.shuffle(randIndx)
        Data, Target = Data[randIndx], Target[randIndx]
        trainData, trainTarget = Data[:3500], Target[:3500]
        validData, validTarget = Data[3500:3600], Target[3500:3600]
        testData, testTarget = Data[3600:], Target[3600:]
    return trainData, validData, testData, trainTarget, validTarget, testTarget

# This website is used to guide me write the code for Logistic Regression using TensorFlow
# [https://www.kaggle.com/autuanliuyc/logistic-regression-with-tensorflow],[https://www.geeksforgeeks.org/ml-logistic-regression-using-tensorflow/],
# [https://web.stanford.edu/class/cs20si/2017/lectures/notes_03.pdf], [https://www.ritchieng.com/machine-learning/deep-learning/tensorflow/regularization/]
def buildGraph(beta1=None, beta2=None, epsilon=None, lossType=None, learning_rate=None):
    # Define parameters for the model
    β = 0
    graph = tf.Graph()
    batch_size = 1750 #change the batch_size to either (100, 700, 1750)
    num_epochs = 700
    with graph.as_default():
        # Input data. For the training data, we use a placeholder that will be fed
        # at run time with a training minibatch.
```

```
        # Declare the variables that need to be learned and initialized
 (weight and bias tensors)
        w = tf.Variable(tf.truncated_normal(shape=(784, 1), mean=0.0, st
ddev=0.5, dtype=tf.float32, seed =None, name=None))
        b = tf.Variable(tf.zeros(1))
        # Define placeholders
        x = tf.placeholder(tf.float32, shape=(batch_size, 784))
        y = tf.placeholder(tf.float32, shape=(batch_size, 1))
        validation_dataset = tf.placeholder(tf.float32, shape=(100, 784
))
        validation_labels = tf.placeholder(tf.int32, shape=(100, 1))
        testing_dataset = tf.placeholder(tf.float32, shape=(145, 784))
        testing_labels = tf.placeholder(tf.int32, shape=(145, 1))
        tf.set_random_seed(521)
        if lossType == "CE Loss Function":
            # Training Computation
            logit = tf.matmul(x, w) + b
            train_prediction = tf.sigmoid(logit)
            loss = tf.losses.sigmoid_cross_entropy(y, train_prediction)
            # Loss function using L2 Regularization
            regularizer = tf.nn.l2_loss(w)
            loss = loss + β/2.0 * regularizer
            # Define the Optimizer
            #optimizer = tf.train.GradientDescentOptimizer(0.001).minimi
ze(loss) #using gradient descent with learning rate of 0.001 to minimize
loss
            optimizer = tf.train.AdamOptimizer(0.001).minimize(loss)
            # Validation Computation
            logit = tf.matmul(validation_dataset,w) + b
            valid_prediction = tf.sigmoid(tf.matmul(validation_dataset,
w) + b)
            valid_loss = tf.losses.sigmoid_cross_entropy(validation_labe
ls, valid_prediction)
            # Loss function using L2 Regularization
            regularizer = tf.nn.l2_loss(w)
            valid_loss = valid_loss + β/2.0 * regularizer
            # Testing Computation
            logit = tf.matmul(testing_dataset,w) + b
            test_prediction = tf.sigmoid(tf.matmul(testing_dataset, w) +
b)
            test_loss = tf.losses.sigmoid_cross_entropy(testing_labels,
test_prediction)
            # Loss function using L2 Regularization
            regularizer = tf.nn.l2_loss(w)
            test_loss = test_loss + β/2.0 * regularizer
    return weight, bias, (predictions>=0.5), trainTarget, l, optimizer,
regularizer
```

2. Implementing Stochastic Gradient Descent [5 pts.]

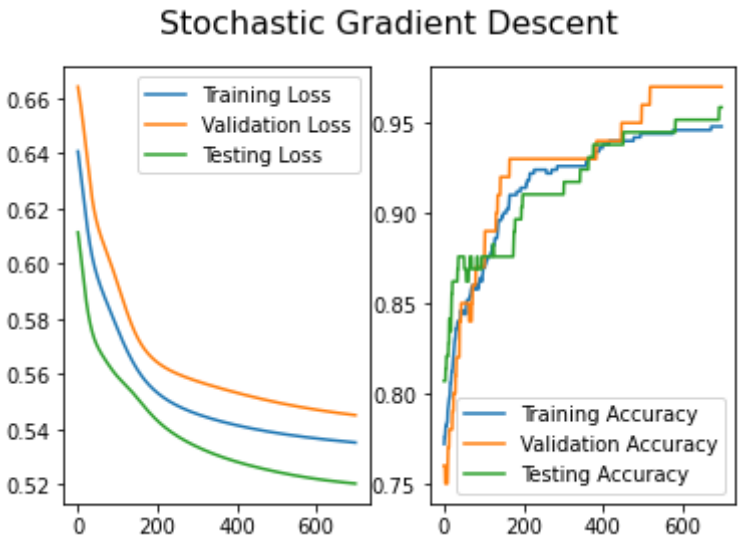
For the training loop, implement the SGD algorithm using a minibatch size of 500 optimizing over 700 epochs. Calculate the total number of batches required by dividing the number of training instances by the minibatch size. After each epoch you will need to reshuffle the training data and start sampling from the beginning again. Initially, set $\lambda = 0$ and continue to use the same α value (i.e. 0.001). After each epoch, store the training and validation losses and accuracies. Use these to plot the loss and accuracy curves.

In [ ]:
```
        # Define the Accuracy Function [https://www.ritchieng.com/machin
e-learning/deep-learning/tensorflow/regularization/]
        # The default threshold is 0.5, rounded off directly
        # Uisng this website for the def accuracy function [https://www.
ritchieng.com/machine-learning/deep-learning/tensorflow/regularization/]
        def accuracy(predictions, labels):
            return (np.sum((predictions>=0.5)==labels) / np.shape(predic
tions)[0])
        with tf.Session(graph=graph) as session:
            num_batches = int(3500/batch_size)
            tf.global_variables_initializer().run()
            training_loss = []
            validating_loss = []
            testing_loss = []
            train_accur = []
            valid_accur = []
            test_accur = []
            for i in range(num_epochs):
                # Input data. For the training data, we use a placeholde
r that will be fed
                # at run time with a training minibatch.
                trainData, validData, testData, trainTarget, validTarget
, testTarget = loadData()
                trainData = trainData.reshape((trainData.shape[0], train
Data.shape[1]*trainData.shape[2]))
                validData = validData.reshape((-1,validData.shape[1]*val
idData.shape[2]))
                testData = testData.reshape((-1,testData.shape[1]*testDa
ta.shape[2]))
            # Run the computations. We tell .run() that we want to run t
he optimizer,
            # and get the loss value and the training predictions return
ed as numpy
            # arrays [using this website: https://www.ritchieng.com/mach
ine-learning/deep-learning/tensorflow/regularization/]
                for j in range(num_batches):
                  minibatch_x = trainData[j*batch_size:(j+1)*batch_size
,]
                  minibatch_y = trainTarget[j*batch_size:(j+1)*batch_siz
e,]
                  feed_dict = {x: minibatch_x,y: minibatch_y, validation
_dataset: validData, validation_labels: validTarget, testing_dataset: te
stData,testing_labels: testTarget}
                  _, trained_w, trained_b, l, predictions, validation_lo
ss, validation_prediction, t_loss, testing_prediction = session.run(
                  [optimizer, w, b, loss, train_prediction, valid_loss,
valid_prediction, test_loss, test_prediction],
                  feed_dict=feed_dict)
                if (i % 1 == 0):
                    training_loss.append(l)
                    validating_loss.append(validation_loss)
                    testing_loss.append(t_loss)
                    train_accur.append(accuracy(predictions, minibatch_y
))
                    valid_accur.append(accuracy(validation_prediction, v
alidTarget))
```

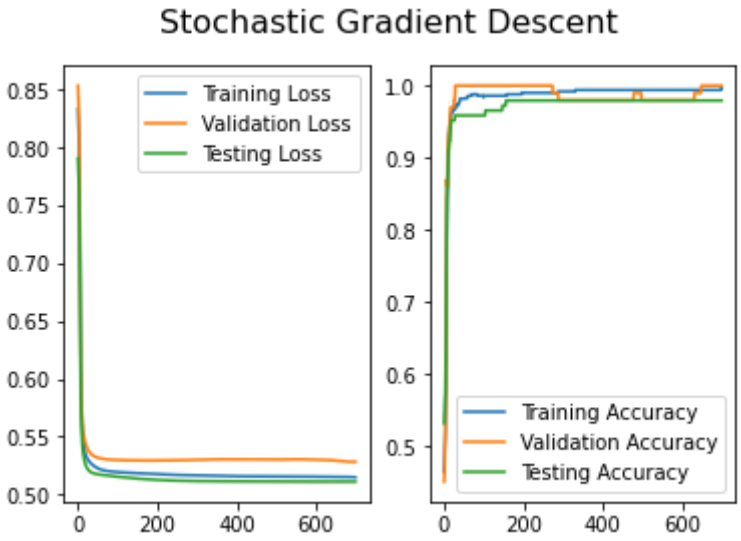
```
                    test_accur.append(accuracy(testing_prediction, testT
arget))
        plt.subplot(121)
        plt.plot(range(num_epochs),training_loss)
        plt.plot(range(num_epochs),validating_loss)
        plt.plot(range(num_epochs),testing_loss)
        plt.legend(['Training Loss', 'Validation Loss', 'Testing Loss'],
loc='upper right')
        plt.subplot(122)
        plt.plot(range(num_epochs),train_accur)
        plt.plot(range(num_epochs),valid_accur)
        plt.plot(range(num_epochs),test_accur)
        plt.suptitle('Adam by optimizing the model using batch sizes of
 B = %s' %batch_size, fontsize=16)
        plt.legend(['Training Accuracy', 'Validation Accuracy', 'Testing
Accuracy'], loc='lower right')
        plt.show()

    return w, b, (predictions>=0.5), trainTarget, l, optimizer, regulari
zer

if __name__ == '__main__':
    w, b, predictions, trainTarget, l, optimizer, regularizer = buildGra
ph(lossType = "CE Loss Function")
```

Comparing the two results, it shows that using Adam by optimizing, the model is this close to converge and the values for accuracy and loss are higher than the other model without Adam

Using "optimizer = tf.train.GradientDescentOptimizer(0.001).minimize(loss)" Not Adam



Using "optimizer = optimizer = tf.train.AdamOptimizer(0.001)" for Adam

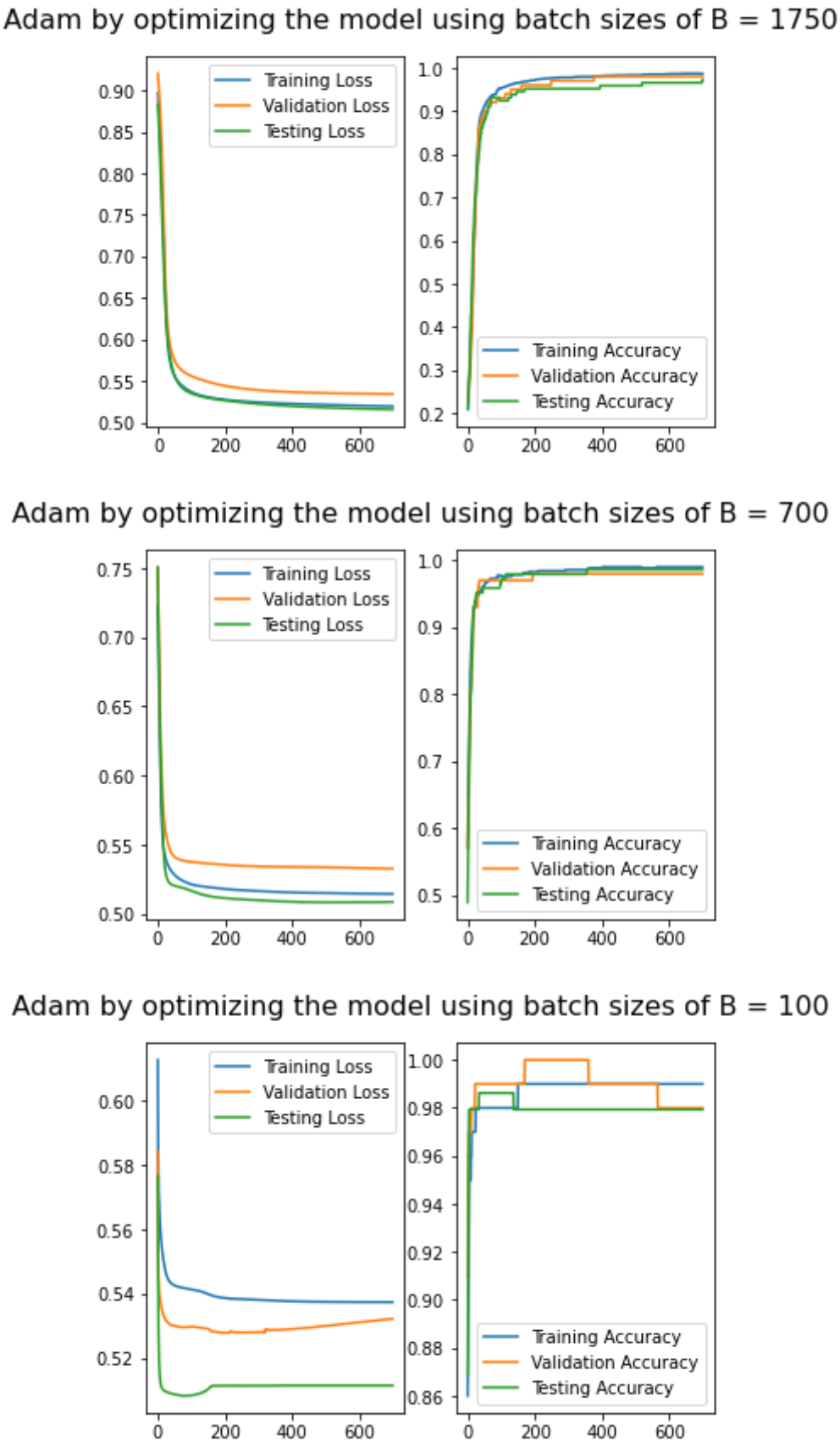


3. Batch Size Investigation [4 pts.]

Study the effects of batch size on behaviour of Adam by optimizing the model using batch sizes of B = {100, 700, 1750}. Also, set $\lambda = 0$ and continue to use $\alpha = 0.001$. For each batch size, plot training/validation loss in one plot and training/validation accuracy in another plot (you need to have a total of 6 plots for this section). What is the impact of batch size on the final classification accuracy for each of the 3 cases? Can you justify this observation?

Answer: The Figure below shows the loss and accuracy changing curves for different batch sizes (100, 700, 1750). On the accuracy curves, the variations between different batch sizes can be seen more easily. As the batch size increased, the produced models decrease in training, validation, and testing datasets. This is because in the manner of applying additional noise to the gradient calculations, mini batch has the regularizing effects on the model.

Figure 3: Using the Adam Batch size for each set with λ = 0 and α = 0.001



4. Hyperparameter Investigation [4 pts.]

Experiment with the following Adam hyperparameters and for each, report on the final training, validation and test accuracies. Explain which value you pick for hyperparameters in each part, and what accuracy you report.

(a) $\beta 1 = \{0.95, 0.99\}$

(b) $\beta 2 = \{0.99, 0.9999\}$

(c) $\in = \{1e - 09, 1e - 4\}$

For this part, use a minibatch size B = 500, a learning rate of $\alpha = 0.001$ with regularization, and optimize over 700 epochs. For each of the three hyperparameters listed above, keep the other two as the default TensorFlow initialization. Note that in order to set β1, β2, and , you may wish to add these parameters to build_graph() inputs.

The code below represents the use of Adam hyperparameters "optimizer = tf.train.AdamOptimizer(learning_rate=0.001, beta1=0.95, beta2=0.99, epsilon=1e-09).minimize(loss)" using this line of code

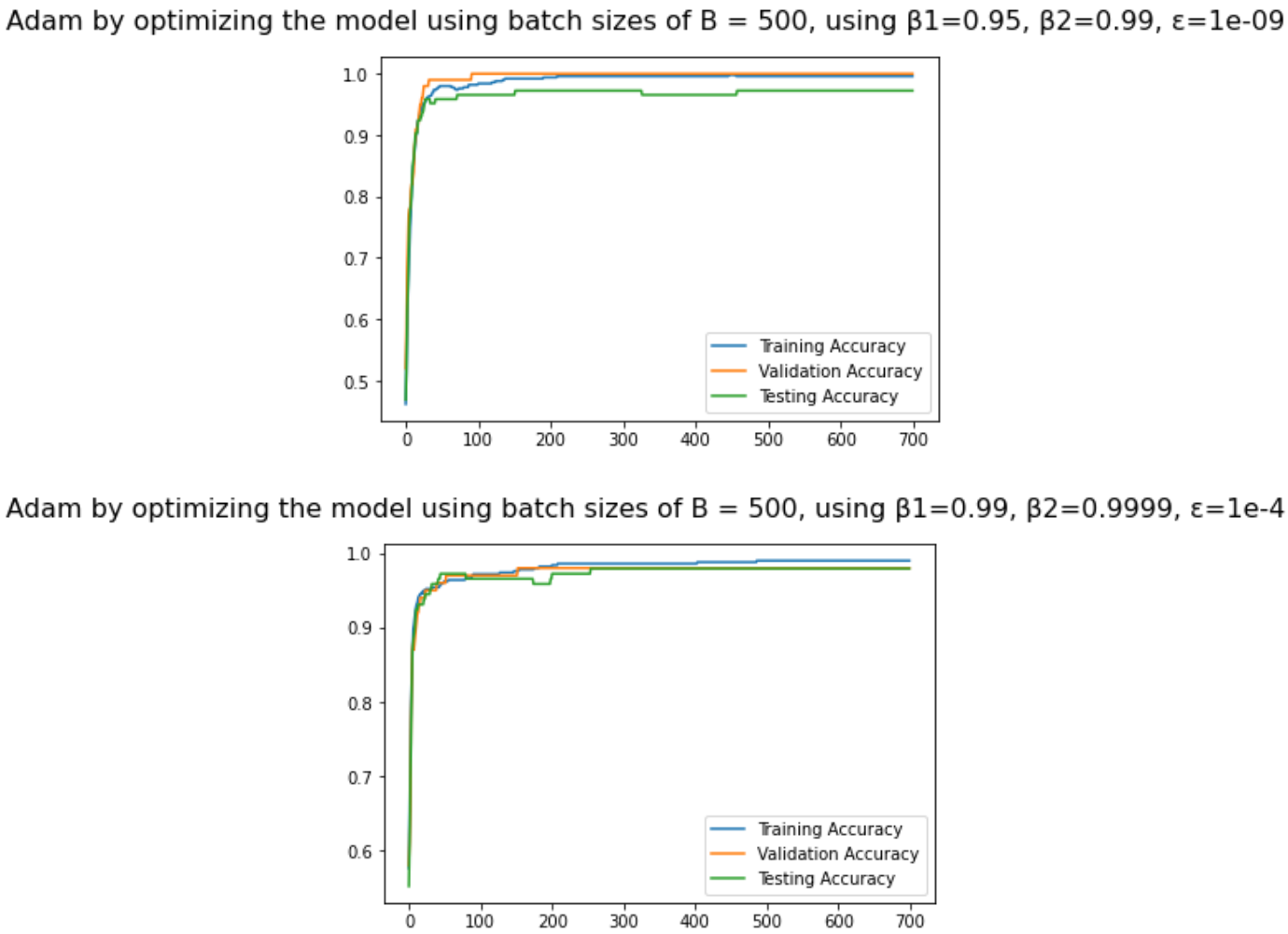
In [ ]:
```
# This website is used to guide me write the code for Logistic Regressio
n using TensorFlow
# [https://www.kaggle.com/autuanliuyc/logistic-regression-with-tensorflo
w],[https://www.geeksforgeeks.org/ml-logistic-regression-using-tensorflo
w/],
# [https://web.stanford.edu/class/cs20si/2017/lectures/notes_03.pdf], [h
ttps://www.ritchieng.com/machine-learning/deep-learning/tensorflow/regul
arization/]
def buildGraph(beta1=None, beta2=None, epsilon=None, lossType=None, lear
ning_rate=None):
    # Define parameters for the model
    β = 0
    graph = tf.Graph()
    batch_size = 500 #change the batch_size to either (100, 700, 1750)
    num_epochs = 700
    with graph.as_default():
        # Input data. For the training data, we use a placeholder that w
ill be fed
        # at run time with a training minibatch.
        # Declare the variables that need to be learned and initialized
 (weight and bias tensors)
        w = tf.Variable(tf.truncated_normal(shape=(784, 1), mean=0.0, st
ddev=0.5, dtype=tf.float32, seed =None, name=None))
        b = tf.Variable(tf.zeros(1))
        # Define placeholders
        x = tf.placeholder(tf.float32, shape=(batch_size, 784))
        y = tf.placeholder(tf.float32, shape=(batch_size, 1))
        validation_dataset = tf.placeholder(tf.float32, shape=(100, 784
))
        validation_labels = tf.placeholder(tf.int32, shape=(100, 1))
        testing_dataset = tf.placeholder(tf.float32, shape=(145, 784))
        testing_labels = tf.placeholder(tf.int32, shape=(145, 1))
        tf.set_random_seed(521)
        if lossType == "CE Loss Function":
            # Training Computation
            logit = tf.matmul(x, w) + b
            train_prediction = tf.sigmoid(logit)
            loss = tf.losses.sigmoid_cross_entropy(y, train_prediction)
            # Define the Optimizer
            #optimizer = tf.train.GradientDescentOptimizer(0.01).minimiz
e(loss) #using gradient descent with learning rate of 0.001 to minimize
 loss
            #optimizer = tf.train.AdamOptimizer(0.001).minimize(loss)
            optimizer = tf.train.AdamOptimizer(learning_rate=0.001, beta
1=0.95, beta2=0.99, epsilon=1e-09).minimize(loss)
            #optimizer = tf.train.AdamOptimizer(learning_rate=0.001, bet
a1=0.99, beta2=0.9999, epsilon=1e-4).minimize(loss)
            # Validation Computation
            logit = tf.matmul(validation_dataset,w) + b
            valid_prediction = tf.sigmoid(tf.matmul(validation_dataset,
w) + b)
            valid_loss = tf.losses.sigmoid_cross_entropy(validation_labe
ls, valid_prediction)
            # Testing Computation
            logit = tf.matmul(testing_dataset,w) + b
            test_prediction = tf.sigmoid(tf.matmul(testing_dataset, w) +
```

```
b)
            test_loss = tf.losses.sigmoid_cross_entropy(testing_labels,
test_prediction)

        # Define the Accuracy Function [https://www.ritchieng.com/machin
e-learning/deep-learning/tensorflow/regularization/]
        # The default threshold is 0.5, rounded off directly
        # Uisng this website for the def accuracy function [https://www.
ritchieng.com/machine-learning/deep-learning/tensorflow/regularization/]
        def accuracy(predictions, labels):
            return (np.sum((predictions>=0.5)==labels) / np.shape(predic
tions)[0])
        with tf.Session(graph=graph) as session:
            num_batches = int(3500/batch_size)
            tf.global_variables_initializer().run()
            training_loss = []
            validating_loss = []
            testing_loss = []
            train_accur = []
            valid_accur = []
            test_accur = []
            for i in range(num_epochs):
                # Input data. For the training data, we use a placeholde
r that will be fed
                # at run time with a training minibatch.
                trainData, validData, testData, trainTarget, validTarget
, testTarget = loadData()
                trainData = trainData.reshape((trainData.shape[0], train
Data.shape[1]*trainData.shape[2]))
                validData = validData.reshape((-1,validData.shape[1]*val
idData.shape[2]))
                testData = testData.reshape((-1,testData.shape[1]*testDa
ta.shape[2]))
            # Run the computations. We tell .run() that we want to run t
he optimizer,
            # and get the loss value and the training predictions return
ed as numpy
            # arrays [using this website: https://www.ritchieng.com/mach
ine-learning/deep-learning/tensorflow/regularization/]
                for j in range(num_batches):
                  minibatch_x = trainData[j*batch_size:(j+1)*batch_size
,]
                  minibatch_y = trainTarget[j*batch_size:(j+1)*batch_siz
e,]
                  feed_dict = {x: minibatch_x,y: minibatch_y, validation
_dataset: validData, validation_labels: validTarget, testing_dataset: te
stData,testing_labels: testTarget}
                  _, trained_w, trained_b, l, predictions, validation_lo
ss, validation_prediction, t_loss, testing_prediction = session.run(
                  [optimizer, w, b, loss, train_prediction, valid_loss,
valid_prediction, test_loss, test_prediction],
                  feed_dict=feed_dict)
                if (i % 1 == 0):
                    training_loss.append(l)
                    validating_loss.append(validation_loss)
                    testing_loss.append(t_loss)
                    train_accur.append(accuracy(predictions, minibatch_y
```

```
))
                    valid_accur.append(accuracy(validation_prediction, v
alidTarget))
                    test_accur.append(accuracy(testing_prediction, testT
arget))
        plt.subplot(111)
        plt.plot(range(num_epochs),train_accur)
        plt.plot(range(num_epochs),valid_accur)
        plt.plot(range(num_epochs),test_accur)
        plt.suptitle('Adam by optimizing the model using batch sizes of
 B = %s, using β1=0.95, β2=0.99, ε=1e-09 ' %batch_size, fontsize=16)
        plt.legend(['Training Accuracy', 'Validation Accuracy', 'Testing
Accuracy'], loc='lower right')
        plt.show()

    return w, b, (predictions>=0.5), trainTarget, l, optimizer

if __name__ == '__main__':
    w, b, predictions, trainTarget, l, optimizer = buildGraph(lossType =
"CE Loss Function")
```

Answer: As shown, the overall accuracy for all three data sets is exactly the same regardless of all values, but the curves for ϵ = 1e-09 have a broad oscillation at the beginning of the curve, while the curves for ϵ = 1e-04 appear to be smoother at the beginning. This was caused by the zero initialise moment vectors, as there was fewer influence of ϵ on weight parameters when ϵ = 1e-09 (very small number), so there were oscillations at the beginning of the curves when zero initialization had been taken into account.

Figure 4: Adam using Epsilon and Beta



5. Comparison against Batch GD [2 pts.]

Comment on the overall performance of the SGD algorithm with Adam vs. the batch gradient descent algorithm you implemented earlier by comparing plots of the losses and accuracies of the Adam vs. batch gradient descent.

Answer: Because of smaller batch sizes and the Adam optimizer effect, the overall output with Adam for training model using SGD algorithm is much better than using the decent batch gradient algorithm I had implemented in part 1. There are several potential reasons why SGD with Adam delivers a greater overall value from the viewpoint of Adam. First, when using smaller batches, opposed to using the whole data set to calculate, the computation time would be much quicker. Finally, it could lead to a quicker convergence rate by using a more precise phase path towards minimum value with the use of the Adam optimizer.