ML Assignment 1

***I have submitted 4 files. A pdf file for the report, a ipynb file for my part 2, a pdf file of the ipynb file and a starter.py for my part 1. Thanks!**

1 Logistic Regression with Numpy

1. Loss function:

$$L = \frac{1}{N}\sum [-y_n log(\sigma(w^T x + b)) - (1 - y_n) log(1 - \sigma(w^T x + b))] + \frac{\lambda}{2}||w||_2^2$$

Gradient of the loss function with respect to w: $\nabla L = \frac{[\sigma(w^T x + b) - y]x_n}{N}$

Gradient of the loss function with respect to b: $\nabla L = \frac{[\sigma(w^T x + b) - y]}{N}$

```
def sigmoid(z):
    return 1/(1+np.exp(-z))

def loss(W, b, x, y, reg):
    n = np.shape(y)[0]
    z = np.dot(x,W) + b
    lce = -1/n*np.sum(y*np.log(sigmoid(z)) + (1-y)*np.log(1-sigmoid(z)))
    lw = (reg/2)*(np.linalg.norm(W)**2)
    return lce+lw

def grad_loss(W, b, x, y, reg):
    n = np.shape(y)[0]
    z = np.dot(x,W)+b
    gradW = np.dot(np.transpose(x),(sigmoid(z)-y))/n + reg*W
    gradB = np.sum(sigmoid(z)-y)/n
    return gradW, gradB
```

Figure 1. Code for the three functions: sigmoid, loss, and grad_loss

```
def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol):
    for i in range(epochs):
        gradW, gradB = grad_loss(W, b, x, y, reg)
        newW = W - alpha*gradW
        newB = b - alpha*gradB
        if np.linalg.norm(W-newW)<error_tol:
            return newW, newB
        else:
            W = newW
            b = newB
    return W,b
```

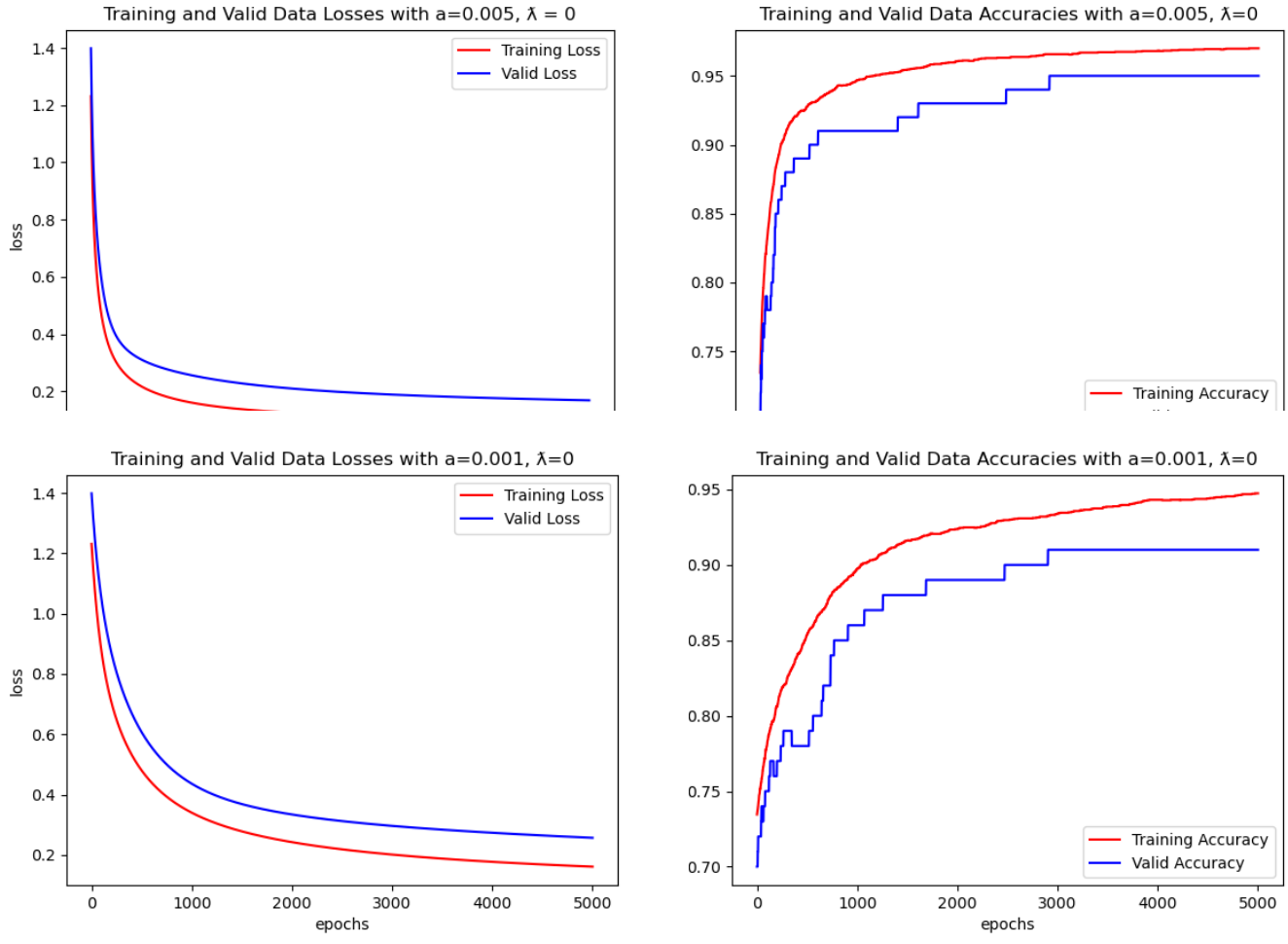
Figure 2. Code for the functions: grad_descent

2. The code for the general gradient descent function (with the original eight inputs) is shown below in Fig.2.
3. For this stage of the analysis, we set the bias to zero, and initialized the weights using Gaussian distribution. Keeping lambda constant at 0, we plotted the the loss and accuracies for the training and validation sets across 5000 epochs with three different learning rates {0.005, 0.001, 0.0001} to investigate its impact. The table below (Table 1) shows the accuracies for all learning rates across all datasets.

	α = 0.005	α = 0.001	α = 0.0001
Training Accuracy	97%	94.7%	85.3%
Valid Accuracy	95%	91%	78%
Test Accuracy	97.93%	97.2%	88.3%

Table 1. Training, validation, and testing accuracies across different learning rates

From the graphs (Figure 3) and the table we can conclude that over 5000 epochs, the best learning rate is α = 0.005 with accuracy of almost 98% on the testing data. This learning rate also results in a faster convergence, and the minimum loss. On the other hand, α = 0.0001 has the lowest accuracy (88%), slowest convergence, and ends with the most loss.



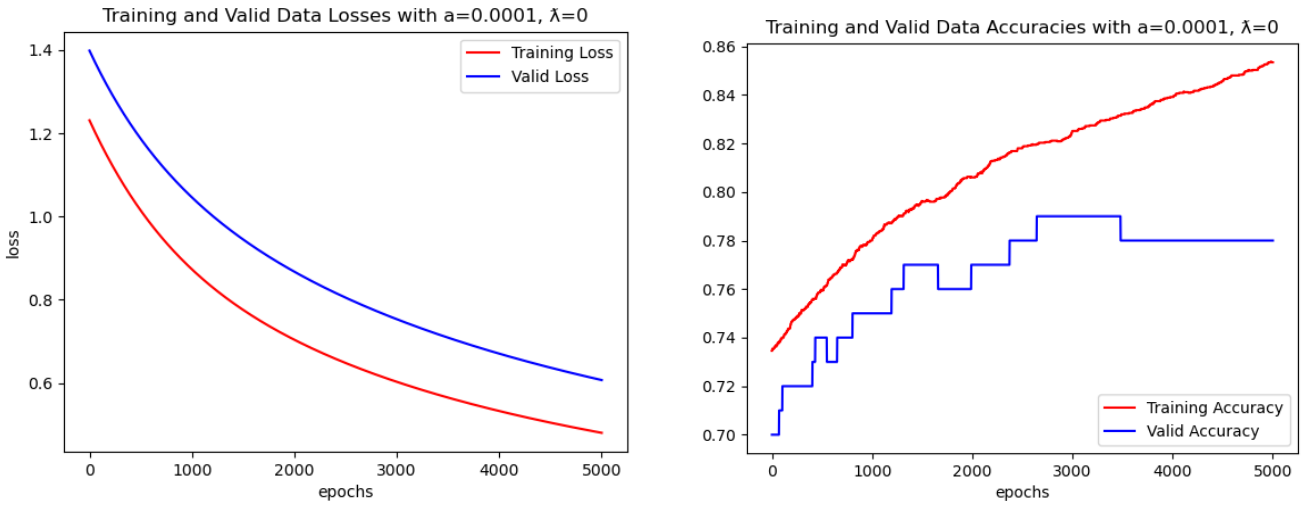


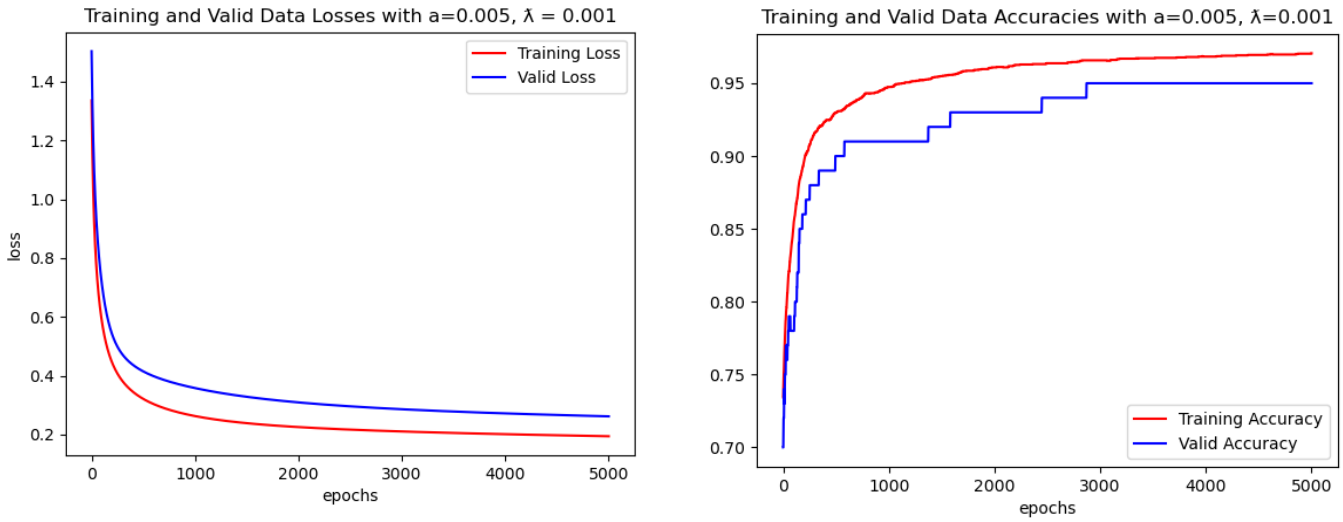
Figure 3. Plots of training and validation losses and accuracies across different learning rates

4. In this step we keep the learning rate constant at α=0.005, while trying different regularization parameters {0.001, 0.1, 0.5} to investigate its effect on learning. The weights, bias, and epochs are kept the same as before. The table below summarizes the accuracies for all datasets across all regularization parameters (Table 2).

	ƛ=0.001	ƛ=0.1	ƛ=0.5
Training Accuracy	97.06%	97.94%	97.7%
Valid Accuracy	95%	97%	97%
Test Accuracy	97.93%	98.62%	96.55%

Table 2. Training, validation, and testing accuracies across different regularization parameters

If we look back at the data for α=0.005 in the last part, we see that the training data reports higher accuracy than the validation data which is an indication of overfitting. The regularization parameter however, can avoid overfitting. Looking at table 3, we see that ƛ=0.1 results in the highest accuracy of almost 99% for the testing data. However, ƛ=0.5 converges faster than ƛ=0.1. Bu overall we can identify the best regularizer as ƛ=0.1 for our set up.



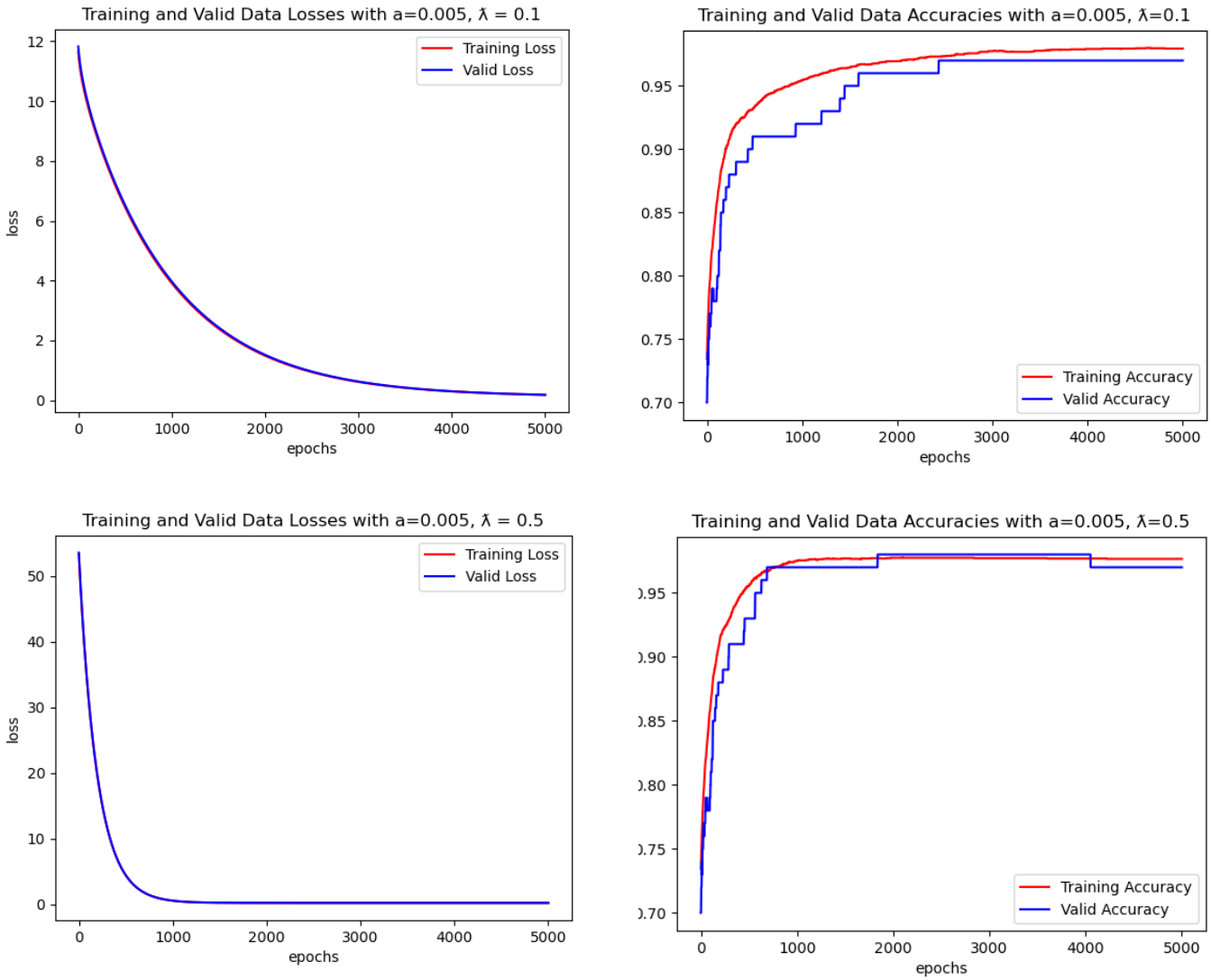


Figure 4. Plots of training and validation losses and accuracies across different regularization parameters

2 Logistic Regression in Tensorflow

1. The code for the general buildGraph() function is shown below in Fig.4

```
def buildGraph(minibatch):
  w = tf.Variable(tf.random.truncated_normal((784, 1),0.0,0.5))
  b = tf.Variable(0.0)
  l = 0


  x = tf.placeholder(tf.float32, (minibatch, 784), name='X')
  y = tf.placeholder(tf.float32, (minibatch, 1), name='Y')

  z = tf.matmul(x,w) + b
  x_pred = tf.sigmoid(z)
  x_loss = tf.losses.sigmoid_cross_entropy(y, x_pred)
  reg = tf.nn.l2_loss(w)
  x_loss += l/2*reg

  optiAndMini = tf.train.AdamOptimizer(0.001).minimize(x_loss)

  return w, b, x, x_pred, y, x_loss, optiAndMini
```

Figure 5. Code for the functions: buildGraph()

2.

```
minibatch = 1750
epochs = 700
batch_num = 3500/minibatch
w, b, x, x_pred, y, x_loss, optiAndMini = buildGraph(minibatch)
l = 0

valid_data = tf.placeholder(tf.float32, (100, 784), name='VD')
valid_target = tf.placeholder(tf.int8, (100, 1), name='VT')

test_data = tf.placeholder(tf.float32, (145, 784), name='TD')
test_target = tf.placeholder(tf.int8, (145, 1), name='TT')

valid_pred = tf.sigmoid(tf.matmul(valid_data, w)+b)
test_pred = tf.sigmoid(tf.matmul(test_data, w)+b)

valid_loss = tf.losses.sigmoid_cross_entropy(valid_target, valid_pred) + l/2*tf.nn.l2_loss(w)
test_loss =  tf.losses.sigmoid_cross_entropy(test_target, test_pred) + l/2*tf.nn.l2_loss(w)

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

x_losses = []
x_accuracy = []

v_losses = []
v_accuracy = []

t_losses = []
t_accuracy = []

trainData, trainTarget, validData, validTarget, testData, testTarget = data()
trainData = trainData.reshape((trainData.shape[0], trainData.shape[1]*trainData.shape[2]))
validData = validData.reshape((validData.shape[0],validData.shape[1]*validData.shape[2]))
testData = testData.reshape((testData.shape[0],testData.shape[1]*testData.shape[2]))

for i in range(epochs):
  index = np.random.permutation(np.shape(trainTarget)[0])
  x_shuffled,y_shuffled = trainData[index], trainTarget[index]
  for batch in range(int(batch_num)):
    xBatch = x_shuffled[i:(i + minibatch),:]
    yBatch = y_shuffled[i:(i + minibatch),:]
    feedDict = {x: xBatch, y: yBatch, valid_data: validData, valid_target: validTarget, test_data: testData, test_target: testTarget}
    _, newW, newB, x_l, x_p, v_l, v_p, t_l, t_p = sess.run(
        [optiAndMini, w, b, x_loss, x_pred, valid_loss, valid_pred, test_loss, test_pred], feed_dict=feedDict)

  x_losses.append(x_l)
  v_losses.append(v_l)
  t_losses.append(t_l)
  x_accuracy.append(accuracy_cal(x_p, yBatch))
  v_accuracy.append(accuracy_cal(v_p, validTarget))
  t_accuracy.append(accuracy_cal(t_p, testTarget))
```

Figure 6. Code for implementing Stochastic Gradient Descent

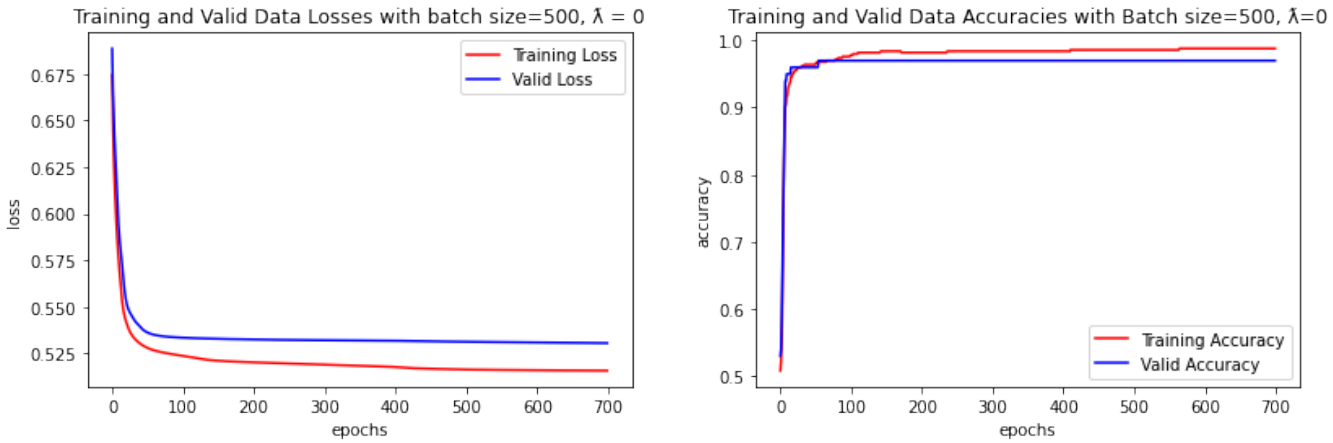


Figure 7. Plots for loss and accuracy of training and valid data with 500 as batch size

3. Keeping the ƛ=0 and α=0.001 constant, we investigated the the batch size effect by trying different batch sizes (100,700,1750). The table below shows the accuracies across all datasets and batch sizes (Table 3). The accuracies don't seem to vary dramatically, but we do see a decrease in the accuracy for B=1750. This batch size also results in a slower convergence. We can conclude that even though small batch sizes don't guarantee reaching the global optima, they have a faster convergence.

	B=100	B=700	B=1750
Training Accuracy	99%	99%	98.5%
Valid Accuracy	99%	98%	98%
Test Accuracy	97.6%	97.2%	95%

Table 3. Training, validation, and testing accuracies across different batch sizes

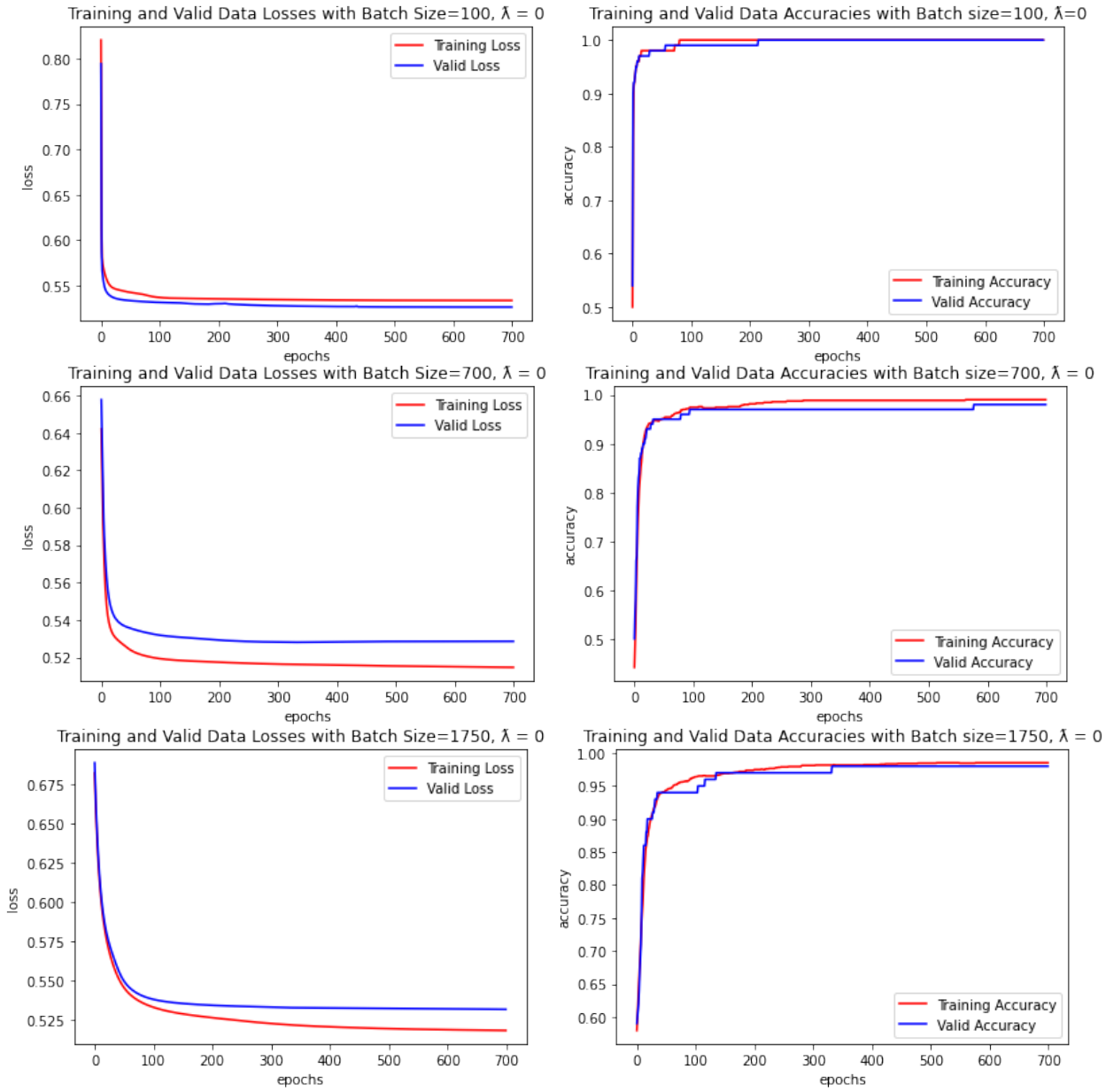


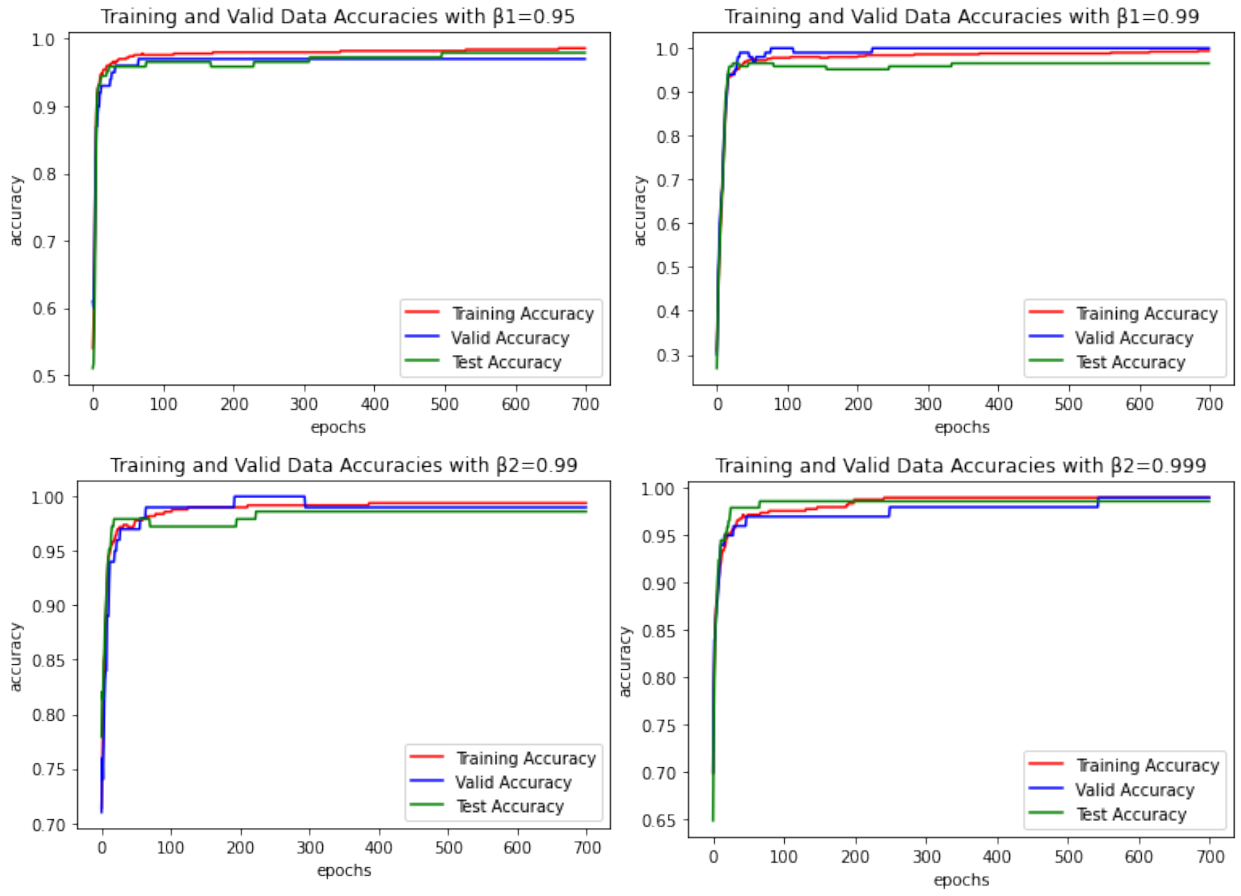
Figure 8. Plots of training and validation losses and accuracies across different batch sizes

4. The table below shows the accuracies across all datasets and hyper parameters (Table 4).

	β1=0.95	β1=0.99	β2=0.99	β2=0.999	ε=1e-09	ε=1e-4
Train Accuracy	98.6%	99.4%	99.4%	99%	99.2%	99.6%
Valid Accuracy	97%	1	99%	99%	99%	99%
Test Accuracy	97.93%	96.55%	98.6%	98.1%	97.9%	97.9%

Table 4. Training, validation, and testing accuracies across different hyper-parameters

A. β1 (0.9 by default) is the exponential decay rate for the first moment estimates and is supposed to speed up the convergence by speeding up the gradient descent. Here, the data shows that with increasing β1 our accuracy decreased and the loss increased. Therefore 0.95 would be a better value for it.
B. β2 (0.999 by default) is the exponential decay rate for the second moment estimates and this too speeds up the convergence. Though the difference in the accuracies between 0.99 and 0.999 are very small, but the table along with the graphs show that 0.99 is the better choice for β2.
C. ε (1e-08 by default) simply prevents division by zero in the optimizer. As the data shows the accuracy isn't impacted by the different ε values and I imagine this is the case as long as the value is very small.



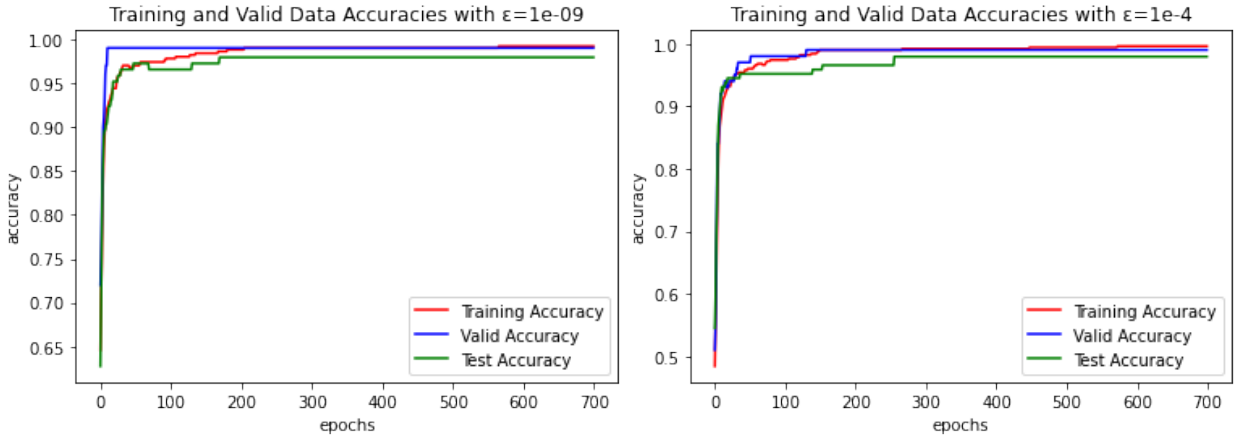


Figure 9. Plots of training and validation losses and accuracies across hyper-parameters

5. While both algorithms with λ=0 and α=0.001 resulted in high accuracies between 95-97.2%, SGD achieved this after only 700 epochs while batch gradient descent took 5000 epochs. After 700 epochs, the SGD loss graph (B=700) has already converged and the loss is less than 0.53. The Batch Gradient Descent however, hasn't converged yet and its loss is about 0.6. In terms of accuracy, after 700 epochs SGC reports 97.2% and BGD reports 83.6%. Overall, the plots shoe that SGC has a faster convergence compare to BGD. Because SGD uses one example at a time, instead of the entire data, it has a noisier path to the minima but it results in significantly fast convergence. SGD is also better for large datasets and, in general, can escape local minima easier. The only disadvantage of SGD is that unlike BGD, it doesn't give the global optima but a good overall solution.