1. Linear Regression

Assumptions:

1. We assumed that data passed into every function, x, was not modified after being returned from loadData(). Thus, we reshaped our data (training, validation or testing) in each function where necessary.
2. W vector was initialized to be of shape = (784, 1).

1. Loss Function and Gradient

We had to reshape our data, x, to suit our implementation. Provided below is the MSE implementation in Python,

```
def MSE(W, b, x, y, reg):
    # reshape y first
    y_reshaped = np.transpose(y)

    # blend the pixel data together per image
    nr_images = x.shape[0]
    x_reshaped = np.reshape(x, (x.shape[0], x.shape[1]*x.shape[2])) # original matrix remains untouched
    x_reshaped = np.transpose(x_reshaped)

    # term1 = means squared error term
    # term2 = weight decay error term
    term1 = np.dot(np.transpose(W), x_reshaped) + b - y_reshaped # matrix multiplication of 2D matrices
    term1 = np.dot(term1, np.transpose(term1)) / nr_images
    term2 = (reg/2 * np.dot(np.transpose(W), W))
    return (term1 + term2)
```

Figure 1: Python Code for MSE Function

Prior to implementing the gradient function in Python, we derived the formula on paper by taking the gradient of the provided final loss function with respect to the weights and bias.

$$Gradient\ with\ respect\ to\ \underline{w} = \frac{2}{N}\sum_{n=1}^{N} (\underline{w}\,.\,\underline{x_n} + b - y_n).\underline{x_n} + \lambda \underline{w}$$

$$Partial\ derivative\ of\ MSE\ with\ respect\ to\ b = \frac{2}{N}\sum_{n=1}^{N} (\underline{w}\,.\,\underline{x_n} + b - y_n)$$

Below is the Python implementation of the above formulas,

```
def gradMSE(W, b, x, y, reg):
    y_reshaped = np.transpose(y)
    nr_images = x.shape[0]
    x_reshaped = np.reshape(x, (x.shape[0], x.shape[1]*x.shape[2])) # original matrix remains untouched
    x_reshaped = np.transpose(x_reshaped)

    term1 = (np.dot(np.transpose(W), x_reshaped) + b - y_reshaped)

    grad_weight = 2/nr_images * np.matmul(x_reshaped, np.transpose(term1)) + reg*W
    grad_bias = 2/nr_images * np.sum(term1)

    return grad_weight, grad_bias
```

Figure 2: Python Code for gradMSE Function

2. Gradient Descent Implementation

Provided below is the implementation of batch gradient descent algorithm in Python based on the above gradients and loss. The function below was modified to accommodate passing of validation and testing data for plotting graphs.

```
def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol, val_data, val_labels, test_data, test_labels):
    # Your implementation here

    difference_weights = 0
    train_error = []
    val_error = []
    test_error = []

    for t in range(epochs):
        grad_w, grad_b = gradMSE(W, b, x, y, reg)
        # full data is sent to gradMSE, means all pictures of 28*28
        vt_w = -grad_w  # set direction for weights to move as negative of gradient
        vt_b = -grad_b
        W_new = W + alpha*vt_w  # update the weights
        b_new = b + alpha*vt_b

        single_train_err = MSE(W_new, b_new, x, y, reg)
        single_val_err = MSE(W_new, b_new, val_data, val_labels, reg)
        single_test_err = MSE(W_new, b_new, test_data, test_labels, reg)
        train_error.append(single_train_err)

        val_error.append(single_val_err)
        test_error.append(single_test_err)

        if (np.linalg.norm(W_new - W) < error_tol): # when difference is under error_tol, then done
            W = W_new
            b = b_new
            train_error_array = np.array(train_error)
            val_error_array = np.array(val_error)
            test_error_array = np.array(test_error)
            plot_errors(train_error_array, val_error_array, test_error_array, t+1, alpha, reg)
            return W_new, b_new
        else:
            W = W_new
            b = b_new

    train_error_array = np.squeeze(np.array(train_error))
    val_error_array = np.squeeze(np.array(val_error))
    test_error_array = np.squeeze(np.array(test_error))
    plot_errors(train_error_array, val_error_array, test_error_array, epochs, alpha, reg)
    return W, b # these are updated final
```

Figure 3: Python Code for grad_descent Function

The above function makes use of additional helper function for plotting. That function merely plots the errors with respect to epochs (plots provided in section 1.3 below).

3. Tuning the Learning Rate

Given below are some plots measuring performance of the above code with varying parameters. For all the plots below, we have used 5000 epochs and $\lambda = 0$. The starting values for weights and bias is taken to be 0.

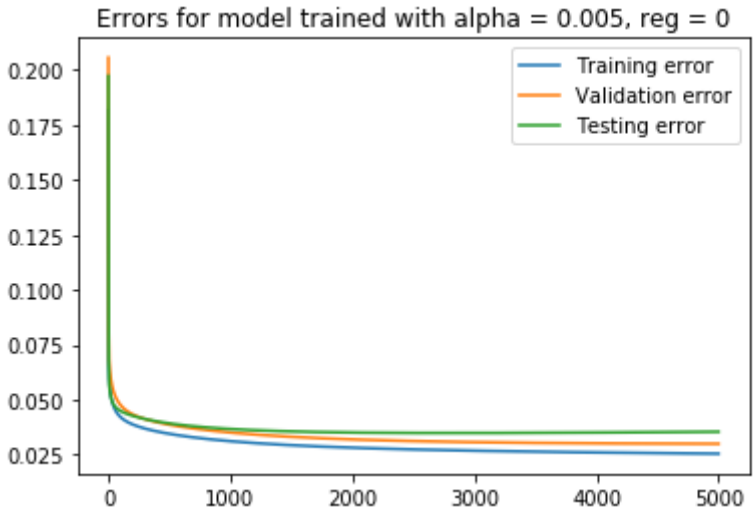


Figure 4: Training, Validation, and Test Losses Plot with learning rate of 0.005, 5000 epochs and $\lambda = 0$

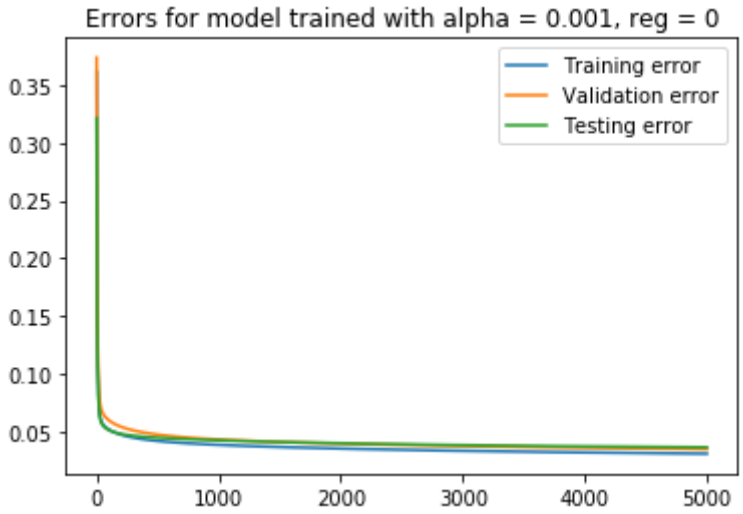


Figure 5: Training, Validation, and Test Losses Plot with learning rate of 0.001, 5000 epochs and $\lambda = 0$

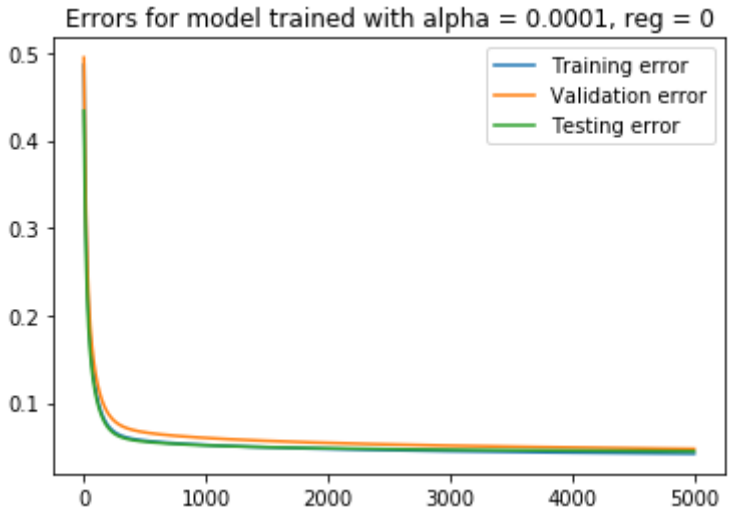


Figure 6: Training, Validation, and Test Losses Plot with learning rate of 0.0001, 5000 epochs and $\lambda = 0$

Numerical statistics from the above plots,

```
alpha = 0.005, reg = 0
Max training error = 0.18157568899911516, Min training error =
0.025384178607605595
Max val error = 0.20565560612721293, Min val error =
0.02989045254598456
Max test error = 0.1972866913543092, Min test error =
0.034794198158235964

alpha = 0.001, reg = 0
Max training error = 0.36107017358425414, Min training error =
0.031078323347745154
Max val error = 0.37350079777412676, Min val error =
0.03501199964299388
Max test error = 0.32114080547124985, Min test error =
0.036605266422231726

alpha = 0.0001, reg = 0
Max training error = 0.486178005663575, Min training error =
0.042092863277804114
Max val error = 0.49464839749975803, Min val error =
0.047585087749007896
Max test error = 0.43383126890396134, Min test error =
0.044859376832925874
```

It can be observed from the above data that decreasing alpha does not impact results in a significant way. Based on the above numerical facts, it can be observed that default alpha value of 0.005 gives the least error values. It also gives the fastest training time as it is the greatest learning rate.

4. Generalization

Given below are the results for investigating the impact of modifying the regularization parameter. In all the instances, we have maintained alpha = 0.005 and 5000 epochs.

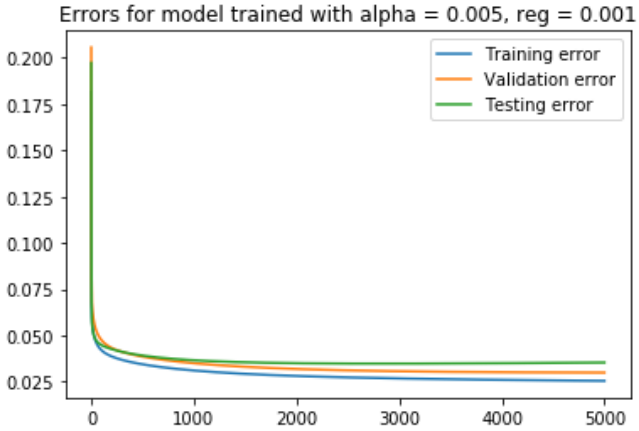


Figure 7: Training, Validation, and Test Losses Plot with learning rate of 0.005, 5000 epochs and $\lambda = 0.001$

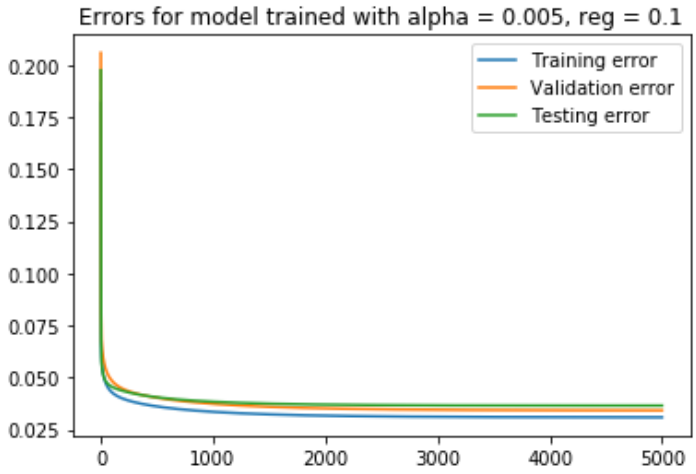


Figure 8: Training, Validation, and Test Losses Plot with learning rate of 0.005, 5000 epochs and $\lambda = 0.1$

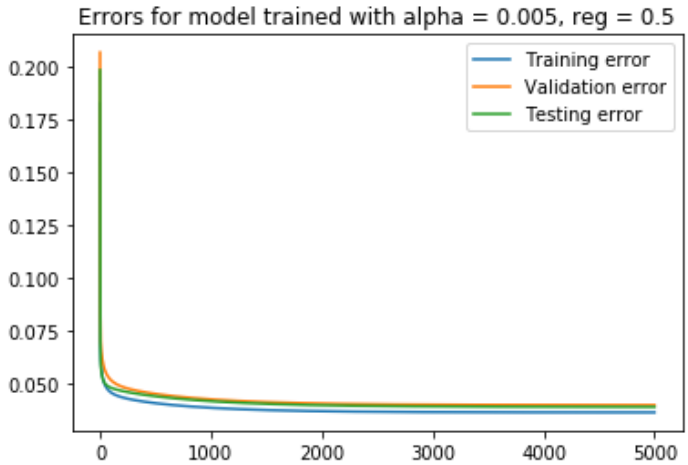


Figure 9: Training, Validation, and Test Losses Plot with learning rate of 0.005, 5000 epochs and $\lambda = 0.5$

Given below are some numerical statistics,

```
alpha = 0.005, reg = 0.001
Max training error = 0.18157767816225853, Min training error =
0.025501465555587497
Max val error = 0.2056575952903563, Min val error = 0.02998195373017605
Max test error = 0.19728868051745257, Min test error =
0.034831310290436505

alpha = 0.005, reg = 0.1
Max training error = 0.18177460531345221, Min training error =
0.030939565894172 02
Max val error = 0.20585452244154998, Min val error =
0.03426907947045441
Max test error = 0.19748560766864626, Min test error =
0.03657275109130344

alpha = 0.005, reg = 0.5
Max training error = 0.1825702705708004, Min training error =
0.03674157976233618
```

```
Max val error = 0.20665018769889817, Min val error =
0.04019690386151807
Max test error = 0.19828127292599446, Min test error =
0.039333158742316035
```

From the above metrics, it can be observed that differences among the error values are insignificant.

5. Comparing Batch GD with normal equation

We know that using least squares formula with zero weight decay,

$$\underline{w_{new}} = (X^T X)^{-1} . X^T . \underline{y}$$

Deriving the value of bias, we have,

$$b_{new} = \frac{1}{N} \sum_{n=1}^{N} (y_n - \underline{w} . \underline{x_n})$$

Below is the implementation in Python,

```python
def leastSquares(x, y):
  y_reshaped = np.transpose(y)
  # blend the pixel data together per image
  nr_images = x.shape[0]
  x_reshaped = np.reshape(x, (x.shape[0], x.shape[1]*x.shape[2])) # original matrix remains untouched
  x_pseudo = np.dot(np.linalg.inv(np.dot(np.transpose(x_reshaped), x_reshaped)), np.transpose(x_reshaped))
  W_new = np.dot(x_pseudo, np.transpose(y_reshaped))

  b_new = (1/nr_images)*np.sum(y-np.dot(x_reshaped, W_new))

  return W_new, b_new
```

Figure 10: Python Code for leastSquare Function

Given below are some metrics comparing performance of least squares with batch GD:

Note: Batch GD is run with 5000 epochs and alpha = 0.005

Type	Training Error	Validation Error	Testing Error	Time Elapsed
Batch GD	0.025384178607605595	0.029890452545984456	0.034794198158235964	12.14537501335144
Normal Equation	0.02300365	0.06067036	0.06680499	0.28224015235900088

Table 1: Performance of Least Squares with Batch GD Comparison

Training, validation and testing errors are comparable and very close. Time elapsed by using batch GD is higher than the least squares implementation.

2. Logistic Regression

2.1 Binary Cross-Entropy Loss

1. Loss Function and Gradient

The cross-entropy loss is defined as:

$$\begin{aligned}\mathcal{L} =& \mathcal{L}_{\mathcal{D}} + \mathcal{L}_W \\ =& \sum_{n=1}^{N} \frac{1}{N} \Big[- y^{(n)} \log \hat{y}(\mathbf{x}^{(n)}) - (1 - y^{(n)}) \log(1 - \hat{y}(\mathbf{x}^{(n)})) \Big] + \frac{\lambda}{2} \|W\|_2^2\end{aligned}$$

We have to reshape x to suit our implementation. Instead of using $W^T x$, we use xW to have an appropriate dimension for matrix multiplication. Provided below is the cross entropy loss implementation in Python.

```
def crossEntropyLoss(W, b, x, y, reg):
    # reshape x
    x = x.reshape([x.shape[0], x.shape[1]*x.shape[2]])

    N = y.shape[0]

    # calculate y_hat from formula given
    y_hat = 1/(1 + np.exp(-(np.matmul(x,W) + b)))

    # calculate cross entropy loss from formula given
    LD = np.sum((-(y*np.log(y_hat))-((1-y)*np.log(1-y_hat)))/N)
    LW = (reg/2)*(np.linalg.norm(W)**2)
    cross_entropy_loss = LD + LW

    return cross_entropy_loss
```

Figure 11: Python Code for crossEntropyLoss Function

Prior to implementing the gradient function in Python, we derived the formula on paper by taking the gradient of the provided cross entropy loss with respect to weights and bias.

Gradient derivation with respect to weights:

$$z = W^T x + b$$

$$\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}}$$

$$J = -ylog(\hat{y}) - (1-y)\log(1-\hat{y})$$

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w}$$

$$\frac{\partial J}{\partial \hat{y}} = \frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}} = \frac{y-\hat{y}}{\hat{y}(1-\hat{y})}$$

$$\frac{\partial \hat{y}}{\partial z} = \frac{\partial \sigma(z)}{\partial z} = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} = \frac{1}{1+e^{-z}} \cdot \frac{(1+e^{-z})-1}{1+e^{-z}}$$

$$= \frac{1}{1+e^{-z}} \cdot (1 - \frac{1}{1+e^{-z}}) = \sigma(z)\big(1-\sigma(z)\big) = \hat{y}(1-\hat{y})$$

$$\frac{\partial z}{\partial w} = x$$

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w} = x^T(\hat{y}-y)$$

$$\mathcal{L}_w = \frac{\lambda}{2}\|W\|^2$$

$$\frac{\partial \mathcal{L}_w}{\partial w} = \lambda\|W\|$$

$$\mathcal{L} = \sum_{n=1}^{N} \frac{1}{N}\Big[-y^{(n)} log\hat{y}\big(x^{(n)}\big) - \big(1-y^{(n)}\big)\log\Big(1-\hat{y}\big(x^{(n)}\big)\Big)\Big] + \frac{\lambda}{2}\|W\|^2$$

$$\frac{\partial \mathcal{L}}{\partial w} = \sum_{n=1}^{N} \frac{1}{N}[x^T(\hat{y}-y)] + \lambda\|W\|$$

Gradient derivation with respect to bias:

$$z = W^T x + b$$

$$\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}}$$

$$J = -y log(\hat{y}) - (1-y)\log(1-\hat{y})$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial b}$$

$$\frac{\partial J}{\partial \hat{y}} = \frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}} = \frac{y-\hat{y}}{\hat{y}(1-\hat{y})}$$

$$\frac{\partial \hat{y}}{\partial z} = \frac{\partial \sigma(z)}{\partial z} = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} = \frac{1}{1+e^{-z}} \cdot \frac{(1+e^{-z})-1}{1+e^{-z}}$$

$$= \frac{1}{1+e^{-z}} \cdot (1 - \frac{1}{1+e^{-z}}) = \sigma(z)\big(1-\sigma(z)\big) = \hat{y}(1-\hat{y})$$

$$\frac{\partial z}{\partial b} = 1$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial b} = (\hat{y} - y)$$

$$\mathcal{L}_w = \frac{\lambda}{2}\|W\|^2$$

$$\frac{\partial \mathcal{L}_w}{\partial b} = 0$$

$$\mathcal{L} = \sum_{n=1}^{N} \frac{1}{N}\Big[-y^{(n)} log\hat{y}\big(x^{(n)}\big) - \big(1-y^{(n)}\big)\log\Big(1-\hat{y}\big(x^{(n)}\big)\Big)\Big] + \frac{\lambda}{2}\|W\|^2$$

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_{n=1}^{N} \frac{1}{N}(\hat{y} - y)$$

Below is the Python implementation of the above formulas.

```python
def gradCE(W, b, x, y, reg):
    # reshape x
    x = x.reshape([x.shape[0], x.shape[1]*x.shape[2]])

    x_transpose = x.transpose()
    N = y.shape[0]

    # calculate y_hat from formula given
    y_hat = 1/(1 + np.exp(-(np.matmul(x,W) + b)))

    gradient_w = np.matmul(x_transpose,(y_hat - y))/N + reg*W
    gradient_b = np.sum((y_hat-y)/N)

    return (gradient_w, gradient_b)
```

Figure 12: Python Code for gradCE Function

2. Learning

Provided below is the implementation of batch gradient descent algorithm in Python based on the above gradients and loss. The function below was modified to accommodate passing of validation and testing data for plotting graphs.

```
def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol, val_data, val_labels, test_data, test_labels, lossType):

    if lossType == "MSE":
      train_error = []
      val_error = []
      test_error = []

      train_accuracy = []
      validity_accuracy = []
      test_accuracy = []

      for t in range(epochs):

          grad_w, grad_b = gradMSE(W, b, x, y, reg)

          vt_w = -grad_w
          vt_b = -grad_b

          W_new = W + alpha*vt_w  # update the weights
          b_new = b + alpha*vt_b

          single_train_err = MSE(W_new, b_new, x, y, reg)
          single_val_err = MSE(W_new, b_new, val_data, val_labels, reg)
          single_test_err = MSE(W_new, b_new, test_data, test_labels, reg)


          train_error.append(single_train_err)
          val_error.append(single_val_err)
          test_error.append(single_test_err)

          train_acc, val_acc, test_acc = accuracy_pt2(W, b, x, y, val_data, val_labels, test_data, test_labels)
```

Figure 13: Python Code for grad_descent Function-1

```
          train_accuracy.append(train_acc)
          validity_accuracy.append(val_acc)
          test_accuracy.append(test_acc)

          if (np.linalg.norm(W_new - W) < error_tol): # when difference is under error_tol, then done
              W = W_new
              b = b_new
              train_error_array = np.array(train_error)
              val_error_array = np.array(val_error)
              test_error_array = np.array(test_error)
              plot_errors(train_error_array, val_error_array, test_error_array, t+1, alpha, reg)
              train_acc_array = np.array(train_accuracy)
              val_acc_array = np.array(validity_accuracy)
              test_acc_array = np.array(test_accuracy)
              plot_accuracy(train_acc_array, validity_acc_array, test_acc_array, t+1, alpha, reg)
              return W_new, b_new
          else:
              W = W_new
              b = b_new

      train_error_array = np.squeeze(np.array(train_error))
      val_error_array = np.squeeze(np.array(val_error))
      test_error_array = np.squeeze(np.array(test_error))
      plot_errors(train_error_array, val_error_array, test_error_array, epochs, alpha, reg)

      train_acc_array = np.squeeze(np.array(train_accuracy))
      validity_acc_array = np.squeeze(np.array(validity_accuracy))
      test_acc_array = np.squeeze(np.array(test_accuracy))
      plot_accuracy(train_acc_array, validity_acc_array, test_acc_array, epochs, alpha, reg)
```

```
elif lossType == "CE":

  train_error = []
  val_error = []
  test_error = []

  train_accuracy = []
  validity_accuracy = []
  test_accuracy = []

  for t in range(epochs):

      grad_w, grad_b = gradCE(W, b, x, y, reg)

      vt_w = -grad_w
      vt_b = -grad_b

      W_new = W + alpha*vt_w  # update the weights
      b_new = b + alpha*vt_b

      single_train_err = crossEntropyLoss(W_new, b_new, x, y, reg)
      single_val_err = crossEntropyLoss(W_new, b_new, val_data, val_labels, reg)
      single_test_err = crossEntropyLoss(W_new, b_new, test_data, test_labels, reg)

      train_error.append(single_train_err)
      val_error.append(single_val_err)
      test_error.append(single_test_err)

      train_acc, val_acc, test_acc = accuracy_pt2(W, b, x, y, val_data, val_labels, test_data, test_labels)

      train_accuracy.append(train_acc)
      validity_accuracy.append(val_acc)
      test_accuracy.append(test_acc)
```

```
      # when difference is under error_tol, then done
      if (np.linalg.norm(W_new - W) < error_tol):
          W = W_new
          b = b_new
          train_error_array = np.array(train_error)
          val_error_array = np.array(val_error)
          test_error_array = np.array(test_error)
          plot_errors(train_error_array, val_error_array, test_error_array, t+1, alpha, reg)
          return W_new, b_new
      else:
          W = W_new
          b = b_new
```

Figure 14: Python Code for grad_descent Function-2

```
  # plot the loss
  plt.plot(train_error)
  plt.plot(val_error)
  plt.plot(test_error)
  plt.title(f'Logistic Regression Loss: alpha = {alpha}, reg = {reg}')
  plt.legend(['Training Loss','Validation Loss','Test Loss'])
  plt.show()

  # plot the accuracy
  plt.plot(train_accuracy)
  plt.plot(validity_accuracy)
  plt.plot(test_accuracy)
  plt.title(f'Logistic Regression Accuracy: alpha = {alpha}, reg = {reg}')
  plt.legend(['Training Accuracy','Validation Accuracy','Test Accuracy'])
  plt.show()

return W, b # these are updated final
```

Figure 15: Python Code for grad_descent Function-3

The above function uses additional helper functions to plot accuracy. The helper function merely compares the difference between the prediction value and training target in order to plot accuracy as shown below.

```python
def accuracy_pt2(W, b, x, y, val_data, val_labels, test_data, test_labels):

    x = x.reshape([x.shape[0], x.shape[1]*x.shape[2]])
    val_data = val_data.reshape([val_data.shape[0], val_data.shape[1]*val_data.shape[2]])
    test_data = test_data.reshape([test_data.shape[0], test_data.shape[1]*test_data.shape[2]])

    y_hat_train = 1/(1 + np.exp(-(np.matmul(x,W) + b)))
    y_hat_val = 1/(1 + np.exp(-(np.matmul(val_data,W) + b)))
    y_hat_test = 1/(1 + np.exp(-(np.matmul(test_data,W) + b)))

    train_acc = np.sum((y_hat_train>=0.5)==y)/(x.shape[0])
    val_acc = np.sum((y_hat_val>=0.5)==val_labels)/(val_labels.shape[0])
    test_acc = np.sum((y_hat_test>=0.5)==test_labels)/(test_data.shape[0])

    return train_acc, val_acc, test_acc
```

Figure 16: Python Code for accuracy_pt2 Function

```python
def plot_errors(d0, d1, d2, epochs, alpha, reg):

    print(f'plot_errors: plotting errors for alpha = {alpha}, reg = {reg}')

    print(f'Max training error = {np.amax(d0)}, Min training error = {np.amin(d0)}')
    print(f'Max val error = {np.amax(d1)}, Min val error = {np.amin(d1)}')
    print(f'Max test error = {np.amax(d2)}, Min test error = {np.amin(d2)}')

    plt.plot(np.arange(epochs), d0, label='Training error')
    plt.plot(np.arange(epochs), d1, label='Validation error')
    plt.plot(np.arange(epochs), d2, label='Testing error')
    plt.title(f'Errors for model trained with alpha = {alpha}, reg = {reg}')
    plt.legend()
    plt.show()
```

Figure 17: Python Code for plot_errors Function

```python
def plot_accuracy(d0, d1, d2, epochs, alpha, reg):

    plt.plot(np.arange(epochs), d0, label='Training Accuracy')
    plt.plot(np.arange(epochs), d1, label='Validation Accuracy')
    plt.plot(np.arange(epochs), d2, label='Testing Accuracy')
    plt.title(f'Logistic Regression Loss: alpha = {alpha}, reg = {reg}')
    plt.legend()
    plt.show()
```

Figure 18: Python Code for plot_accuracy Function

Below is the plot of loss and accuracy curves for training, validation, and test data set.

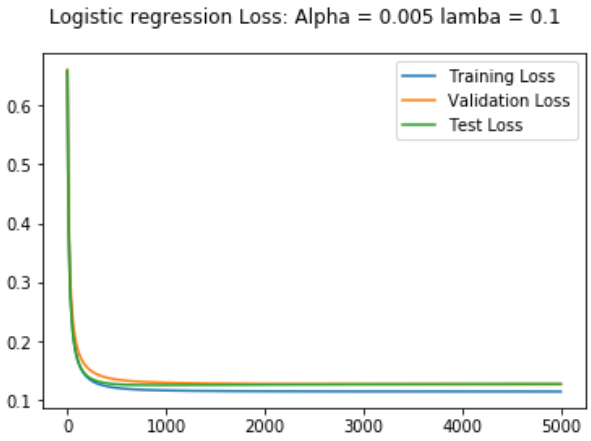


Figure 19: Training, Validation, and Test Plot of Logistic Regression Loss

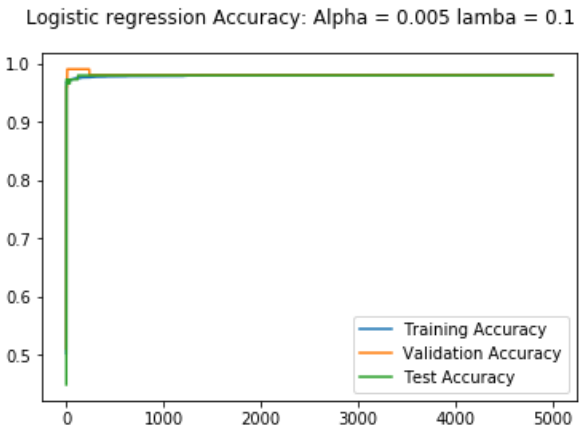


Figure 20: Training, Validation, and Test Plot of Logistic Regression Accuracy

3. Comparison to Linear Regression

The plot below is the comparison between cross entropy loss for logistic regression and MSE loss from linear regression. As the graph shows, logistic regression is faster because cross entropy loss converges slower than MSE loss.

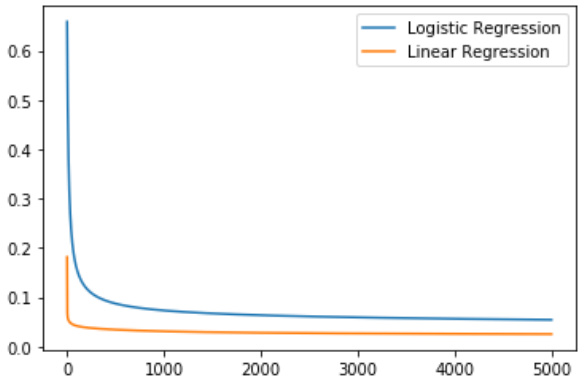


Figure 21: Comparison between Logistic Regression Loss and Linear Regression Loss

3. Batch Gradient Descent vs. SGD and Adam

3.1 SGD

1. Building the Computational Graph

```
def buildGraph(lossType="MSE", batchSize= 500, betaOne=0.9, betaTwo=0.999, learning_rate=0.001, Epsilon= 1e-08):
  #load the training, validation, and test data and targets
  trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()
  beta = 0
  tf.set_random_seed(421)

  #Initialize weight and bias tensors
  ShapeWVector = (trainData.shape[1]*trainData.shape[2])

  W = tf.Variable(tf.truncated_normal(shape = (ShapeWVector, 1), mean = 0.0, stddev = 0.5,
                                      dtype = tf.dtypes.float32, seed = None, name = "weight"))
  bias = tf.zeros(shape=(1), dtype=tf.dtypes.float32)
  X = tf.placeholder(dtype = tf.dtypes.float32, shape = (None, ShapeWVector), name = "X")
  Y = tf.placeholder(dtype = tf.dtypes.float32, shape = (None, 1), name = "Y",)
  myLambda = tf.placeholder(dtype = tf.dtypes.float32, shape = None, name = "myLambda")


  if lossType == "MSE":
    prediction = tf.matmul(X, W) + bias
    loss = tf.losses.mean_squared_error(labels = Y, predictions = prediction)
    regularizer = tf.nn.l2_loss(weight)
    loss += beta*regularizer

  elif lossType == "CE":
    logit = tf.matmul(X, W) + bias
    prediction = tf.sigmoid(logit)
    loss = tf.nn.sigmoid_cross_entropy_with_logits(labels = Y, logits = prediction)
    loss = tf.reduce_mean(loss)
    regularizer = tf.nn.l2_loss(W)
    loss += beta*regularizer

  optimizer = tf.train.AdamOptimizer(learning_rate= learning_rate, beta1= betaOne, beta2= betaTwo, epsilon= Epsilon).minimize(loss)

  #Either add SGD code here, or make a new function!

  return weight, bias, prediction, y, x, loss, optimizer
```

Figure 22: Code for buildgraph Function

2. Implementing Stochastic Gradient Descent

Accuracy function is shown below:

```
def accuracy (x,W,y):
  W = W.eval()
  y_hat = np.matmul(x,W)

  for i in range(len(y_hat)):
      if (y_hat[i][0])>=0.5: y_hat[i][0]=1
      else: y_hat[i][0] = 0

  score = 0
  for i in range(len(y_hat)):
    if y_hat[i][0] == y[i][0]: score+=1

  return score/len(y)
```

Figure 23: Accuracy Function used in SGD Function

SGD function is shown below:

```
def SGD (lossType="MSE", batchSize= 500, betaOne=0.9, betaTwo=0.999, learning_rate=0.001, Epsilon= 1e-08):

  trainData, validData, testData, trainTarget, validTarget, testTarget = loadData()

  weight, bias, prediction, y, x, loss, optimizer= buildGraph(lossType, batchSize, betaOne, betaTwo, learning_rate, Epsilon)

  trainAccuracy = []
  trainLoss = []

  validAccuracy = []
  validLoss = []

  testAccuracy = []
  testLoss = []

  epochs = 700
  numBatches = int(3500/batchSize)
  validData = np.reshape(validData, (100, 784))
  testData = np.reshape(testData, (145, 784))

  with tf.Session() as sess:

    sess.run(tf.global_variables_initializer())
    for j in range(epochs):

      #Shuffling data starts
      temp = np.c_[trainData.reshape(len(trainData), -1), trainTarget.reshape(len(trainTarget), -1)]
      np.random.shuffle(temp)
      trainData = temp[:, :trainData.size//len(trainData)].reshape(trainData.shape)
      trainTarget = temp[:, trainData.size//len(trainData):].reshape(trainTarget.shape)
      #shuffling data ends

      for i in range(numBatches):


        data = trainData[i*batchSize: (i+1)*batchSize]
        target = trainTarget[i*batchSize: (i+1)*batchSize]

        data = np.reshape(data,(batchSize, 784))
        target = np.reshape(target, (batchSize, 1))
        sess.run(optimizer, feed_dict = {x:data, y:target})

      trainLoss.append(sess.run(loss, feed_dict = {x:data, y:target}))
      validLoss.append(sess.run(loss, feed_dict = {x:validData, y:validTarget}))
      testLoss.append(sess.run(loss, feed_dict = {x:testData, y:testTarget}))

      trainAccuracy.append(accuracy(data,weight,target))
      validAccuracy.append(accuracy(validData,weight,validTarget))
      testAccuracy.append(accuracy(testData,weight,testTarget))

    #different plotting scales
    if lossType == "CE":
      LossYlim1 = 0.45
      LossYlim2 = 0.8
      AccuYlim1 = 0.5
      AccuYlim2 = 1.05

    elif lossType == "MSE":
      LossYlim1 = 0
      LossYlim2 = 18
      AccuYlim1 = 0.2
      AccuYlim2 = 1.0

    fig = plt.figure()
    plt.plot(trainAccuracy)
    plt.plot(validAccuracy)
    plt.plot(testAccuracy)
    plt.legend(['Train', 'Valid', 'Test'], loc='lower right')
    plt.xlabel('Epoch', fontsize=16)
    plt.ylabel('Accuracy', fontsize=16)
    axes = plt.gca()
    axes.set_ylim([AccuYlim1, AccuYlim2])
    axes.set_xlim([-50,720])
    plt.show()
```

```
fig = plt.figure()
plt.plot(trainLoss)
plt.plot(validLoss)
plt.plot(testLoss)
plt.legend(['Train', 'Valid', 'Test'], loc='lower right')
plt.xlabel('Epoch', fontsize=16)
plt.ylabel('Loss', fontsize=16)
axes = plt.gca()
axes.set_ylim([LossYlim1,LossYlim2])
axes.set_xlim([-50,720])
plt.show()

print("final training accuracy: ", trainAccuracy[-1])
print("final valid accuracy: ", validAccuracy[-1])
print("final test accuracy: ", testAccuracy[-1])
```

Figure 24: Stochastic Gradient Descent Function

```
SGD(lossType="MSE", batchSize= 500, betaOne=0.9, betaTwo=0.999, learning_rate=0.001, Epsilon= 1e-08)
```

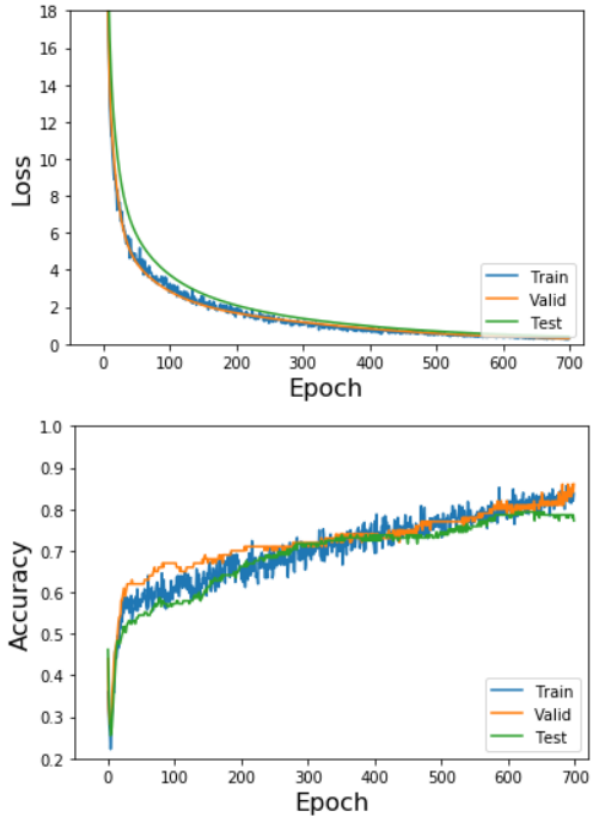
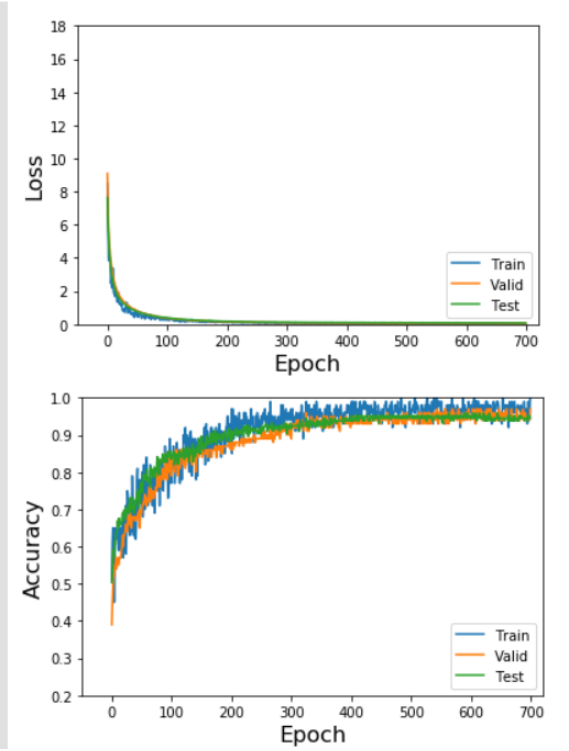


Figure 25: Loss and Accuracy Curves for MSE, batch size = 500, learning rate = 0.001

3. Batch Size Investigation

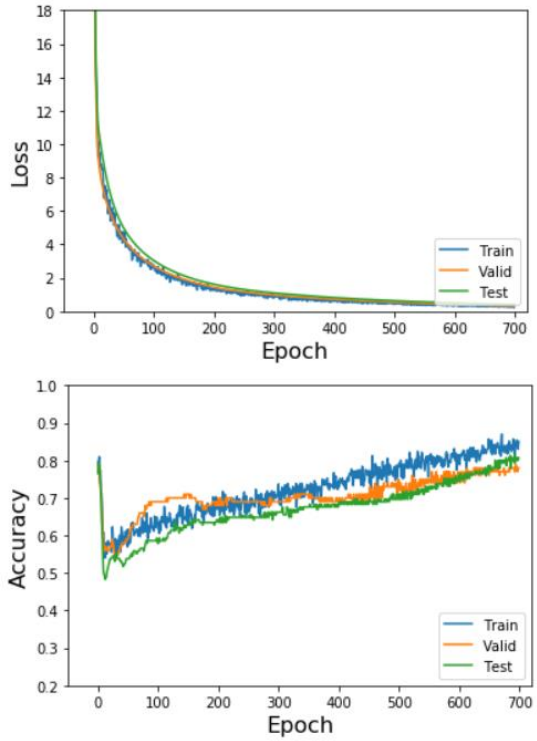
Batch Size= 100:

```
SGD(lossType="MSE", batchSize= 100)
```



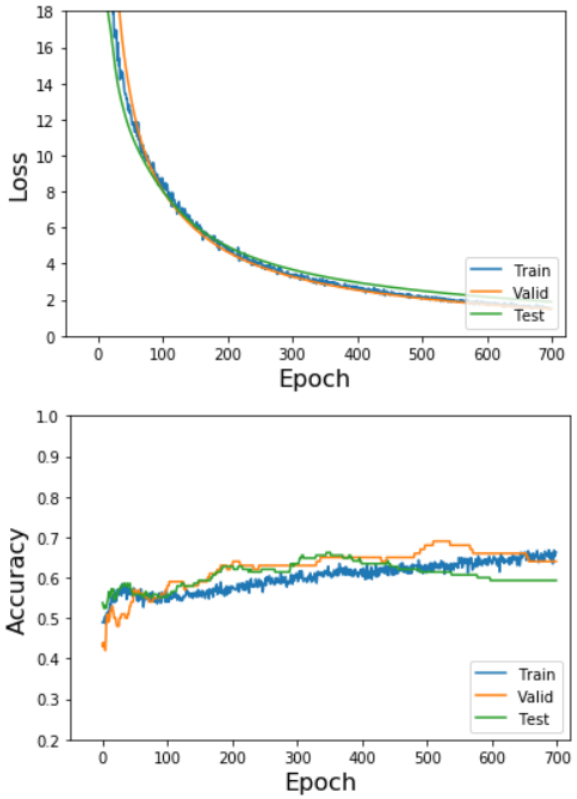
Batch Size= 700:

```
SGD(lossType="MSE", batchSize= 700)
```



Batch Size= 1750:

```
SGD(lossType="MSE", batchSize= 1750)
```



As seen from the figure 26, as the batch size was increased from 100 to 1750, the loss is seen to increase and have a relatively lower rate of decay in all three cases. Further, the accuracy of the graphs decreases with increasing batch size. For a batch size of 100, the accuracy reaches above 95%, whereas with a batch size of 1750, the accuracy remains below 70%.

With a higher batch size, we minimize the gradient more times, resulting in a noisier graph with higher variance, which learns faster and achieves a higher accuracy but takes more time to compute these gradients.

Figure 26: Graphs for MSE loss and accuracy for various batch sizes

4. Hyperparameter Investigation

The following hyper-parameter information is retrieved from tensorflow documentation:

procedure ADAM(f, θ_0 ; α, β_1, β_2)

m_0, v_0, $t \leftarrow [0, 0, 0]$ # Initialize moment estimates
and timestep to zero

Begin optimization procedure

while θ_t has not converged **do**

$t \leftarrow t + 1$ # Update timestep

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ # Compute gradient of objective

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ # Update first moment estimate

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (g_t \odot g_t)$ # Update second moment estimate

$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ # Create unbiased estimate $\widehat{m}_t$

$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ # Create unbiased estimate $\widehat{v}_t$

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \lambda)$ # Update objective parameters

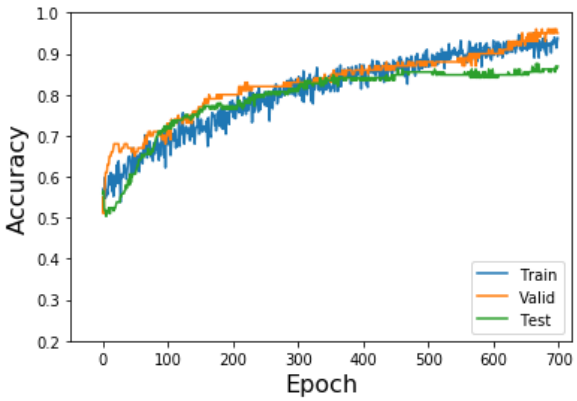
return θ_t # Return final parameters

The final accuracies for training, validation, and testing datasets is shown below for the change in parameters mentioned in the lab:

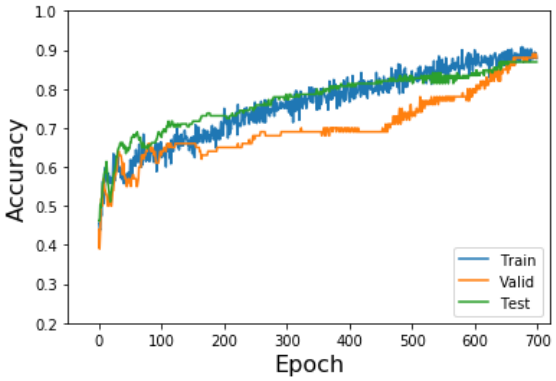
Final Accuracy (%)	**β1= 0.95**	**β1= 0.99**	**β2= 0.99**	**β2= 0.9999**	**ε = 1e-9**	**ε = 1e-4**
Training	93.8	87.8	96	77.2	88.6	93
Validation	95	88	88	72	84	90

Testing	86.9	85.91	92.4	68.27	88.3	88.3

Table 2: Final Accuracy values for various hyper-parameters in MSE



β1= 0.95

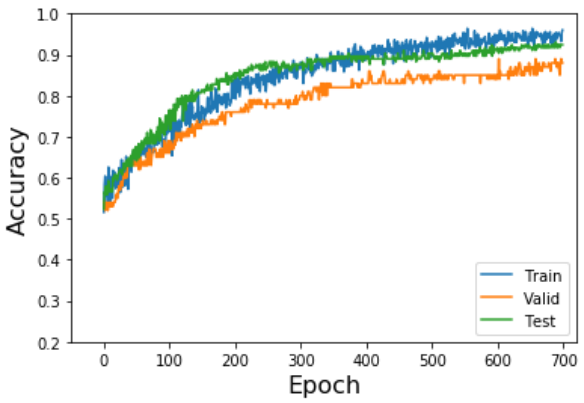


β1= 0.99

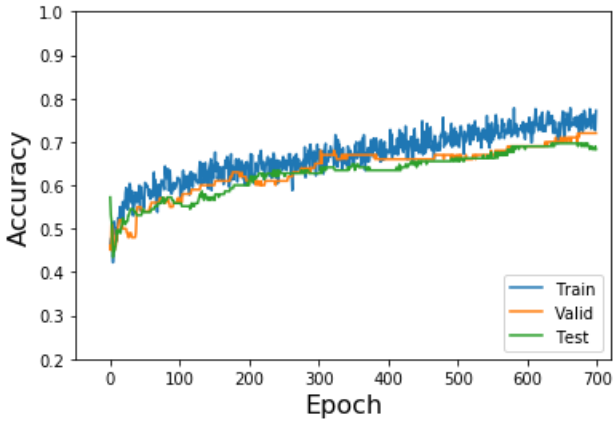
Table 3: Accuracy curves for change of β1

β1 Hyperparameter Analysis:

The adam optimizer provides two beta values with a purpose of speeding up the gradient descent, which makes the model converge faster than the standard gradient descent algorithm. β1 is defined to be the exponential decay rate for the first moment estimates used for incorporating momentum while calculation of gradient descent. As noticed from the graphs above, when β1 = 0.95, there are less oscillations and a higher accuracy is reached after 700 epochs. In contrast, when β1 = 0.99, there are higher oscillations at the beginning in the accuracy curves, and final accuracy values are lower for training, validation, and test datasets. This is because a higher β1 means that more historical values are incorporated into calculating current weight values. This means that the model largely relied on historical values, adding too much momentum and lowering the accuracy.



β2= 0.99

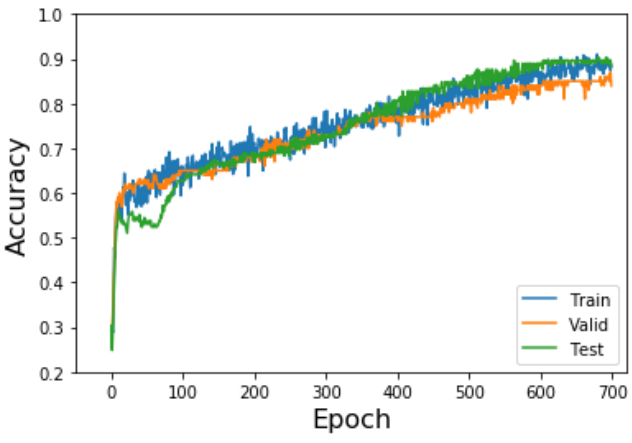


β2= 0.9999

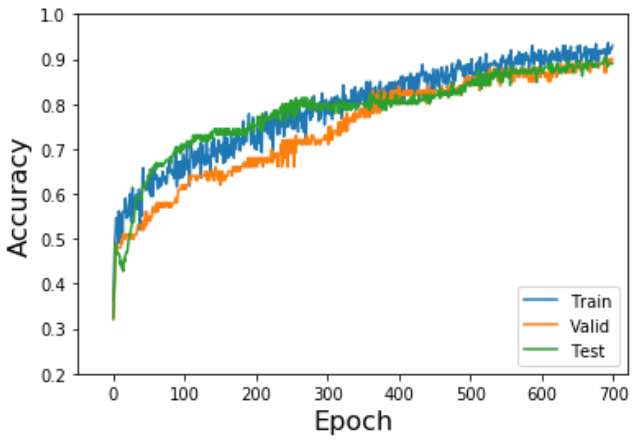
Table 4: Accuracy curves for change of β2

β2 Hyperparameter Analysis:

β2 is defined to be the exponential decay rate for the second moment estimates. It is similar to using Root Mean Squared Propagation when doing optimization. Formally, β2 is used to update the second raw moment estimate (Vt) during optimization. The accuracy was much higher when using β2 = 0.99 as compared to β2 = 0.9999, as noticeable from the plotted accuracy graphs. The final accuracy values for training, validation, and test were also much lower when β2 = 0.9999. As noticeable from the Adam Optimizer equations, when β2 is small, the expression decays to zero faster. When large, too many historical values are taken into consideration when computing weights, leading to choosing the wrong direction for the next step which explains the lower accuracy.



ε = 1e-9



ε = 1e-4

Table 5: Accuracy curves for change of ε

ε Hyperparameter Analysis:

ε is a small number to avoid division with zero during the optimization process of the Adam Optimizer when updating the variable Vt variable when gradient is a small value. By default it is kept to be 1e-08, however, we experiment with it to be ε = 1e-04 and ε = 1e-09. The graphs for accuracy look to be similar with both ε values. The final accuracy values for testing data set depict similar performance for both ε values. The ε value has an effect in the first iteration when first and second moment values are zero. Other than that, unless a major change in ε value is made, the accuracy for model will remain largely unaffected.

5. Cross Entropy Loss Investigation

Graphs for Loss and Accuracy for Cross Entropy with similar parameters as in 3.1.2

```
SGD(lossType="CE")
```

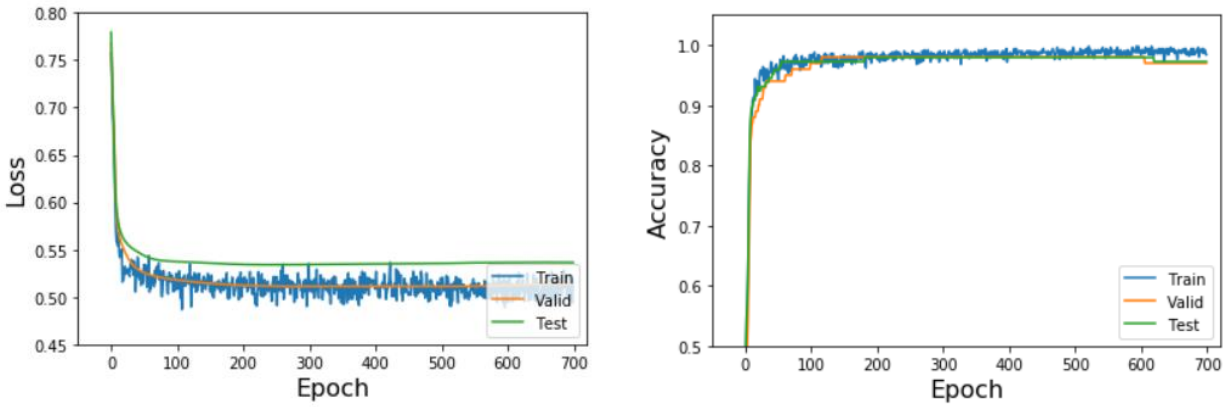
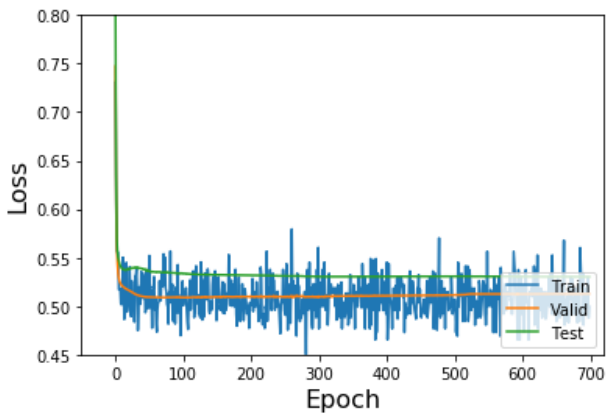
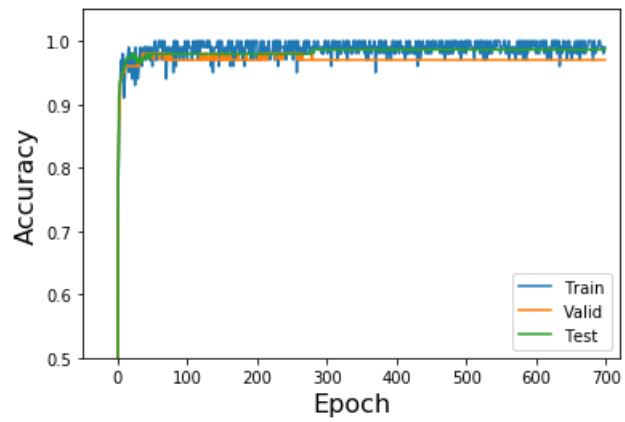


Figure 27: Loss and Accuracy Curves for CE Using Default Hyper-parameters

Accuracy graphs for Loss and Accuracy for Cross Entropy with varying batch sizes as in 3.1.3



Batch Size = 100



Batch Size = 100

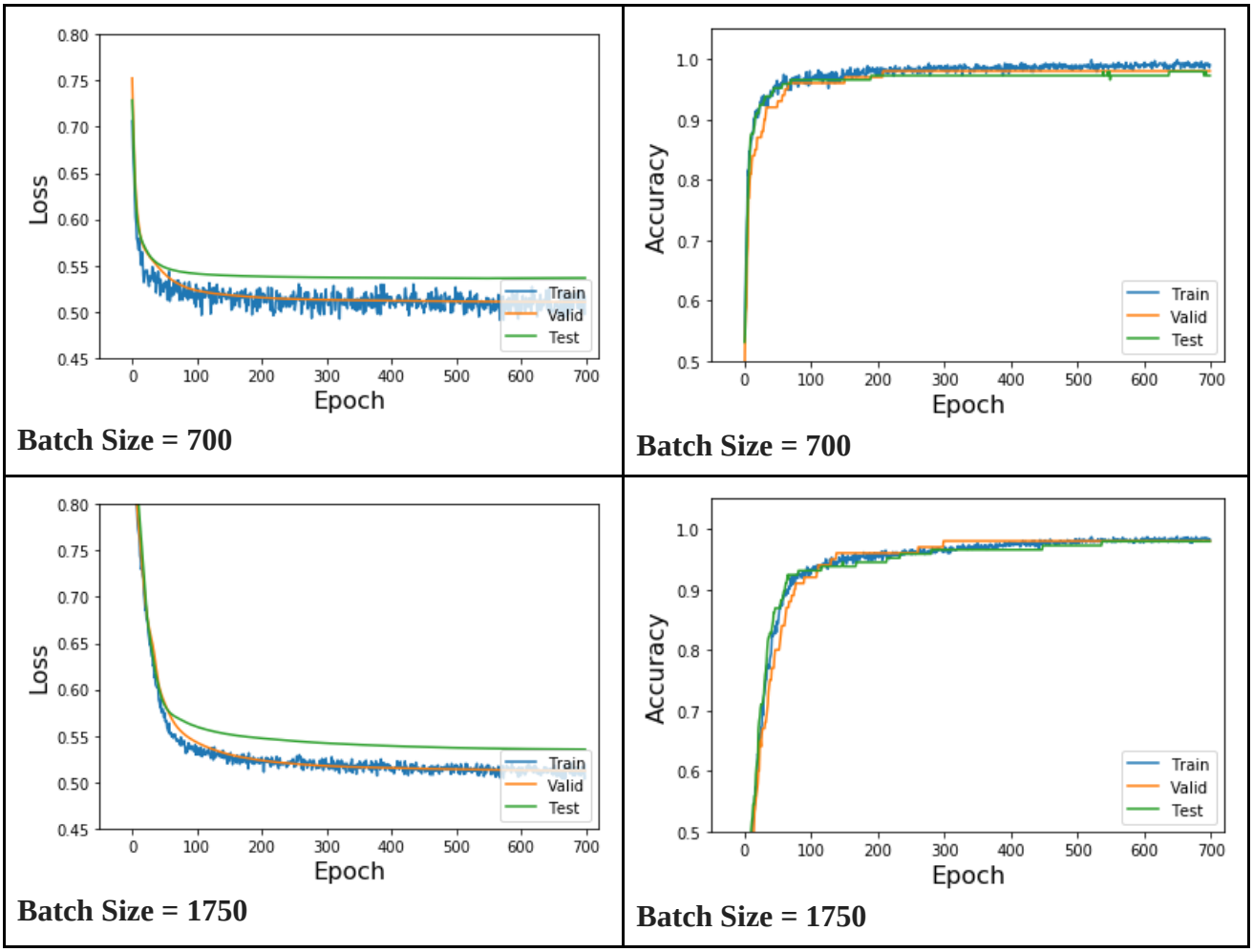


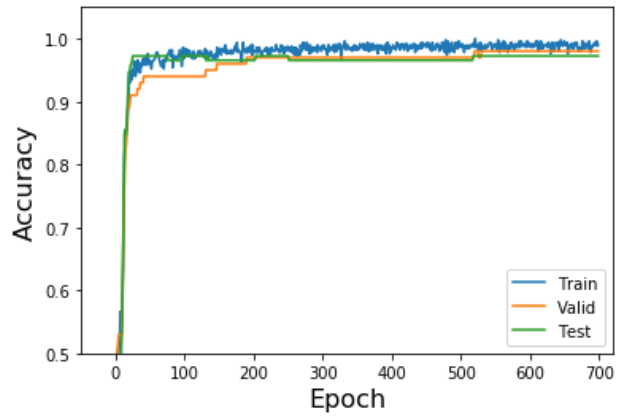
Table 6: Accuracy and Loss curves for CE with varying batch sizes

As noticed in Table 6, the accuracy reduces with increasing batch size. At a batch size of 100, the accuracy for training, validation, and test data sets reaches almost 100% in fewer than 50 epochs. Reasoning is similar to that in MSE, we compute the gradient more times with a smaller batch size, resulting in a noisier graph with higher variance, as noticeable, but it learns much faster.

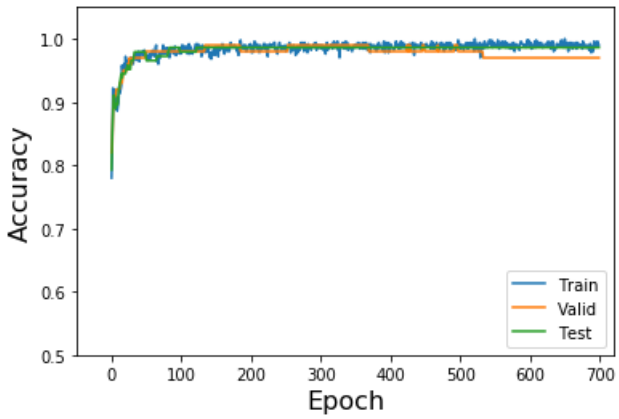
Final Accuracy values and accuracy graphs for CE with varying hyper-parameters as in 3.1.4:

Final Accuracy (%)	**β1= 0.95**	**β1= 0.99**	**β2= 0.99**	**β2= 0.9999**	**ε = 1e-9**	**ε = 1e-4**
Training	99	99	99	98.6	99	99.2
Validation	98	97	97	98	97	98
Testing	97.24	98.62	97.24	97.9	97.93	96.55

Table 7: Final Accuracy Values for Various Hyper-parameters in CE



β1= 0.95

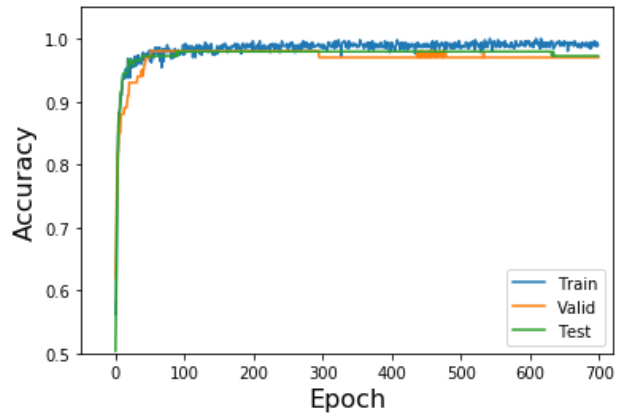


β1= 0.99

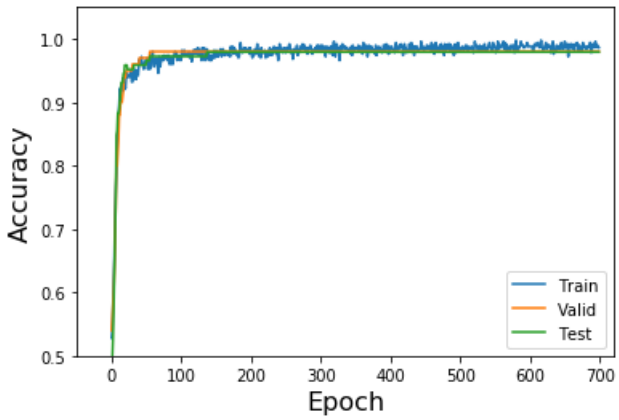
Table 8: Accuracy curves for CE with varying β1 values

β1 Hyperparameter Analysis:

The adam optimizer provides two beta values with a purpose of speeding up the gradient descent, which makes the model converge faster than the standard gradient descent algorithm. β1 is defined to be the exponential decay rate for the first moment estimates used for incorporating momentum while calculation of gradient descent. As noticed from the graphs above, there is not much of a difference with changing β1 values, as they both converge to same accuracy in similar time.



β2= 0.99



β2= 0.9999

Table 9: Accuracy curves for CE with varying β2 values

β2 Hyperparameter Analysis:

β2 is defined to be the exponential decay rate for the second moment estimates. It is similar to using Root Mean Squared Propagation when doing optimization. Formally, β2 is used to update the second raw moment estimate (Vt) during optimization. With β2 = 0.9999, the accuracy converges to slightly slower as compared to β2 = 0.99, and also yields a slightly lower final accuracy compared to β2 = 0.99 as noticed in Table 7. This is caused because of too much

weight given to historical values, slowing the convergence.

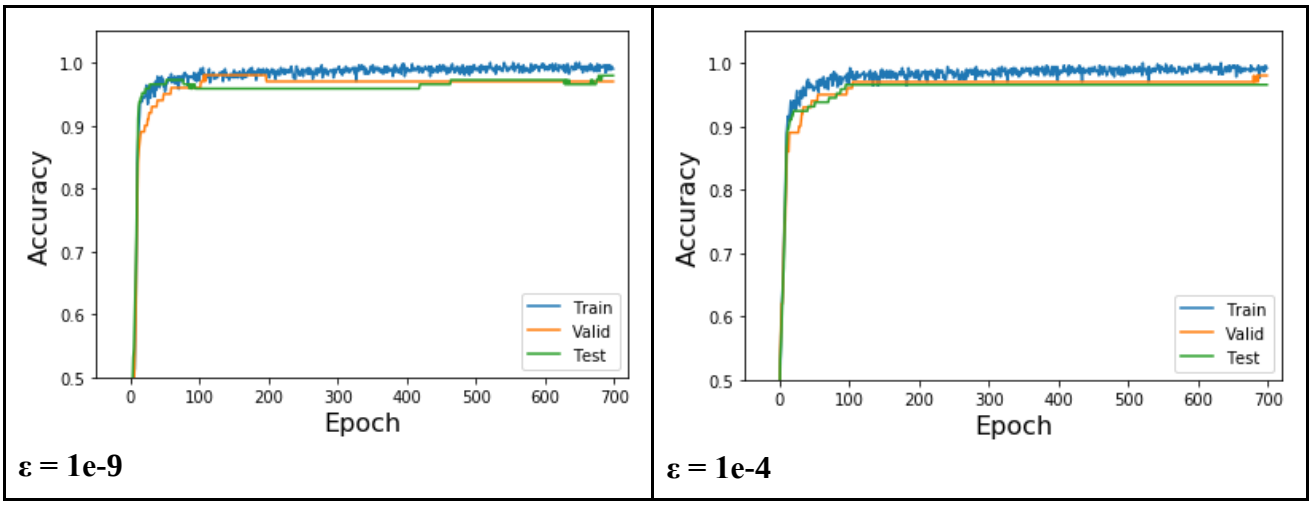


Table 10: Accuracy curves for CE with varying ε values

ε Hyperparameter Analysis:

ε is a small number to avoid division with zero during the optimization process of the Adam Optimizer when updating the variable Vt variable when gradient is a small value. The final accuracy values for testing data set depict similar performance for both ε values. The ε value has an effect in the first iteration when first and second moment values are zero. Other than that, unless a major change in ε value is made, the accuracy for model will remain largely unaffected. This is noticeable from the negligible difference in accuracy final values in Table 7.

Comparison of CE with MSE:

As noticeable in the Loss and Accuracy graphs for Cross Entropy (CE) with default parameters (figure 27), the starting loss is very low compared to MSE loss which is noticeable in figure 25. Further, the decay in loss was much faster in CE than in MSE. Further, upon examining the graphs for accuracy, it is seen that higher accuracy values are reached much earlier in CE than in MSE. The final accuracy values are also higher in CE. The consistency in values was also higher for accuracy curves in train, validation, and test data sets when using CE.

Upon changing the β1, β2, and ε values for CE, it is analyzed that all the final accuracies are higher than those achieved in MSE upon changing the same parameters. Further, changing the hyper-parameters had a minimal effect on the final accuracy values in CE when compared to the relatively large effect exerted in MSE. The CE model can reach a higher accuracy much faster than MSE model, so tuning/optimizing the hyperparameters did not make too much a difference as 700 epochs were already more than enough.

6. Comparison against Batch GD

When analyzing overall performance for the training model, using SDG algorithm with Adam Optimizer is much better than using batch gradient descent algorithm, due smaller batch sizes possible and effects of the Adam Optimizer.

Analysis of the accuracy:

Batch Gradient Descent Algorithm:

- For MSE, the final accuracy only reached around a maximum of 0.65 (computed outside of report requirements in part 1)
- For CE, the graph reached above 95% accuracy after 5000 epochs as seen in figure 20

SGD algorithm with Adam Optimizer:

- Accuracy for MSE reached about 77% on the test data set with default Adam Optimizer hyper-parameters, and was mostly consistent with training and validation data set curves.
- Accuracy for CE reached 97.24% on test data set with default Adam Optimizer hyper-parameters after 700 epochs.

In both the cases (MSE and CE), using the SGD optimizer yields better results of lower number of epochs.

Analysis of the loss:

Batch Gradient Descent Algorithm:

- For MSE, figure 21 shows that the loss almost touches zero after 5000 epochs
- For CE, figure 21 shows that the loss almost touches zero after 5000 epochs

SGD algorithm with Adam Optimizer:

- For MSE, figure 25 shows that the loss starts at 18 and almost touches zero after 700 epochs
- For CE, figure 27 shows that loss starts at around 0.75 and goes to 0.5 after 700 epochs.

In this case, the SGD algorithm for MSE has best performance, as loss decreases to zero in only 700 epochs.

Though some data points may show otherwise, SDG with Adam optimizer proves to provide a better overall performance on the basis of loss, accuracy, and computational speed. Adding momentum when computing weights is something that Adam Optimizer does while also having a faster converge rate. Further, smaller batch sizes lead to a much faster computation. So, the SDG algorithm with Adam Optimizer definitely provides a better overall performance.