ECE 521 Assignment 2

# Work Breakdown

|  |  |
| --- | --- |
| Group Member Name | Contribution Percentage |
| Jixong Deng | 33% |
| Jeffrey Kirman | 33% |
| Connor Smith | 33% |

# Part 1: Linear regression

Linear regression is performed using the following minibatch stochastic gradient descent (SGD) algorithm:

|  |
| --- |
| **Init:** **w** = **0** # Weights vector  **xj** = [1 x­j1 xj2 ... xjd]T # Single in feature  **X** = [**x1** ... **xn**] # Matrix of all in-features  **y** = [**y1** ... **yn**] # Vector of all targets  **Step:** for each epoch  Shuffle **X -> X’** # Randomly shuffle the vectors in **X** into **X’**  for each mini-batch: **X’** = [**x’(mini-batch\_size)\*i** ... **x’(mini-batch\_size)\*(i+1)**]  **gt** = ∇Ein(**w(t)**) # Calculate the gradient of the loss  **w(t+1)** = **w(t)** – η**gt** # Update the weights, where η is the learning factor  **Result:** return **w** |

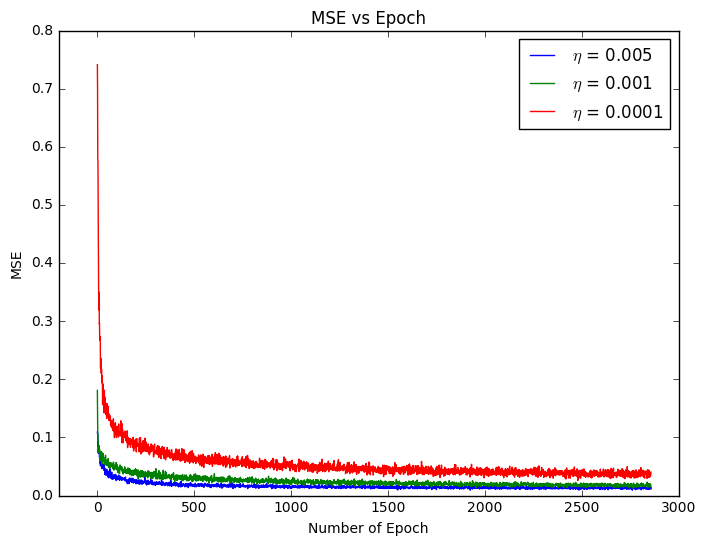
The **SGD** function (**linear\_regression2.py** in *Appendix A*) implements this. This function creates a graph in which with placeholders for input matrix **X** and target vector **y**. It then calculates the MSE loss according to the following equation:

Where λ is the decay constant. The gradients are then calculated using a function in the tensorflow API, tf.gradients. Finally, the weights are updated.

After the graph is created the function calculates the number of epochs based on the iterations and mini-batch size. It then shuffles an array of indices and then uses them to index the matrix of in-features into a mini-batch. feed\_dict is then used to feed the inputs into the placeholders and run the graph for one iteration.

## Part 1 – 1

The results of changing the learning rate (η) can be seen in the figure below. Here we can see that as the learning rate is increased the quicker it converges and the lower final value it converges to. Therefore, **the best η in this case is 0.005.**



## Part 1 – 2: Effect of the mini-batch size

The final values of the MSE loss as well as computation time are summarized in the table below:

|  |  |  |
| --- | --- | --- |
| **Mini-batch Size** | **MSE loss** | **Computation time (seconds)** |
| 500 | 0.0149 | 206 |
| 1500 | 0.0176 | 433 |
| 3500 | 0.0171 | 1243 |

The best mini-batch size in terms of training time is 500. This makes sense since each iteration contains less operations (i.e. less MSE loss and less gradients to compute), hence each update will happen faster.

*Note: for mini-batches that do not equally divide the number of training examples, the remainder of unused examples are used in a shortened mini-batch at the end of each epoch.*

## Part 1 – 3: Generalization

The final values of the accuracy on the validation and test set, as well as computation time are summarized in the table below. When calculating accuracy, if a prediction was above 0.5 it was said to be classified as 1, and below 0.5, classified as 0.

|  |  |  |  |
| --- | --- | --- | --- |
| **λ** | **Validation Accuracy** | **Test Accuracy** | **Computation time (seconds)** |
| 0 | 0.959 | 0.99 | 209 |
| **0.001** | **0.966** | **0.98** | **210** |
| 0.1 | 0.952 | 0.95 | 211 |
| 1 | 0.938 | 0.94 | 211 |

The best decay coefficient in terms of validation accuracy is λ = 0.001. Here the validation accuracy is 0.966 and the test accuracy is 0.98. Using weight decay can help the model by regularizing the data as to not overfit the test set. It is useful in this sense because it gives a tunable parameter to change how the model predicts values outside of the training set.

## Part 1 – 4: Comparing SGD with the normal equation

The normal equation (excluding the decay coefficient) is

Using this we can calculate a prediction for the weights. The results of this can be see in the following table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **MSE loss** | **Validation Accuracy** | **Test Accuracy** | **Computation Time (seconds)** |
| **SGD** | 0.0121 | 0.966 | 1.0 | 234 |
| **Normal Equation** | 0.0094 | 0.958 | 0.97 | 0.0213 |

Here we see that the normal equation performs better in terms of lower MSE but SGD performs in accuracy. In terms of computation time the SGD takes much longer than using the normal equation. That being said, SGD would be the better method to use with exceptionally large datasets since you could lower the batch size and retain high accuracy, whereas calculating matrix multiplications and inverses for the normal equation would become more and more computationally expensive.

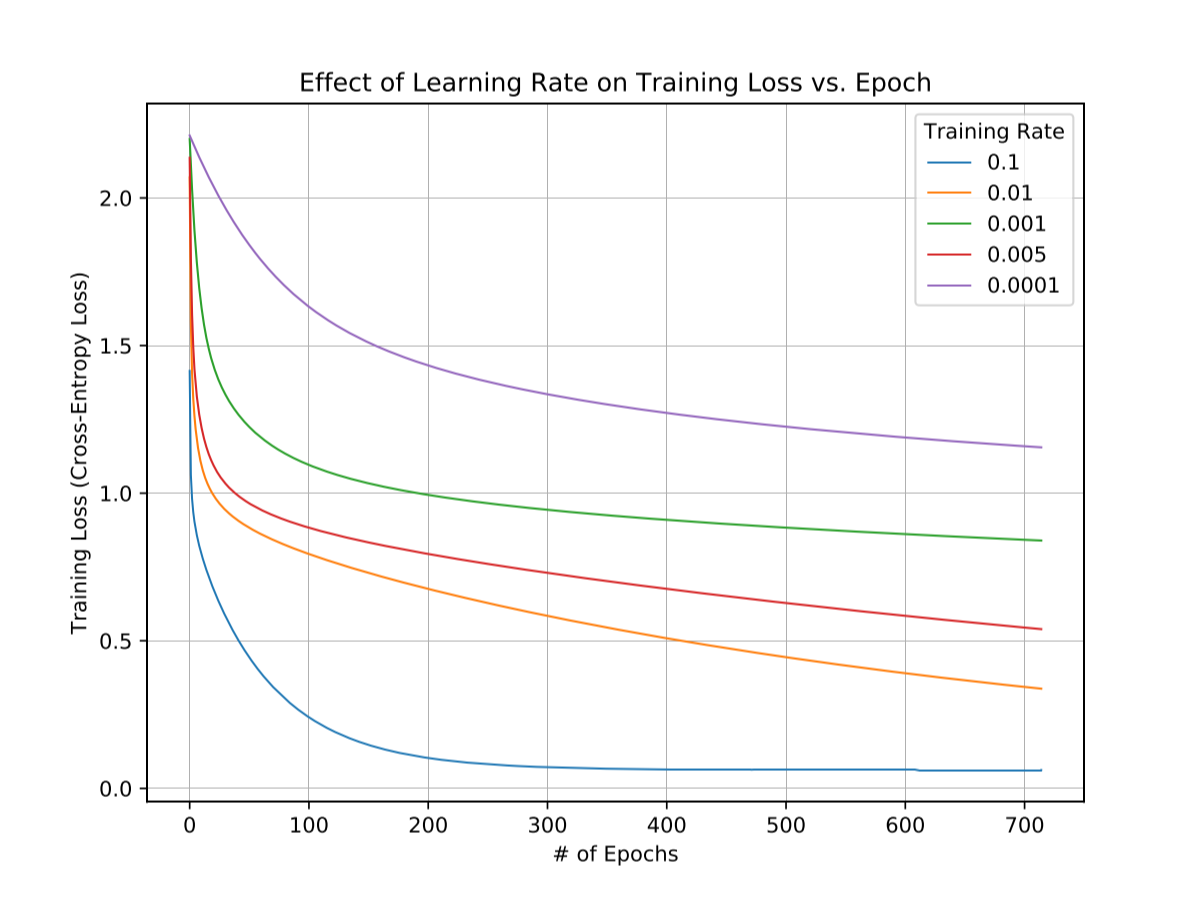
# Part 2: Logistic Regression

The Mean-Squared-Error loss function above is suitable for classic regression tasks, however it can be overly sensitive to mislabelled examples and outliers in the training data for a classification task. As a result, we find that the cross-entropy loss function described below can result in better model performance.

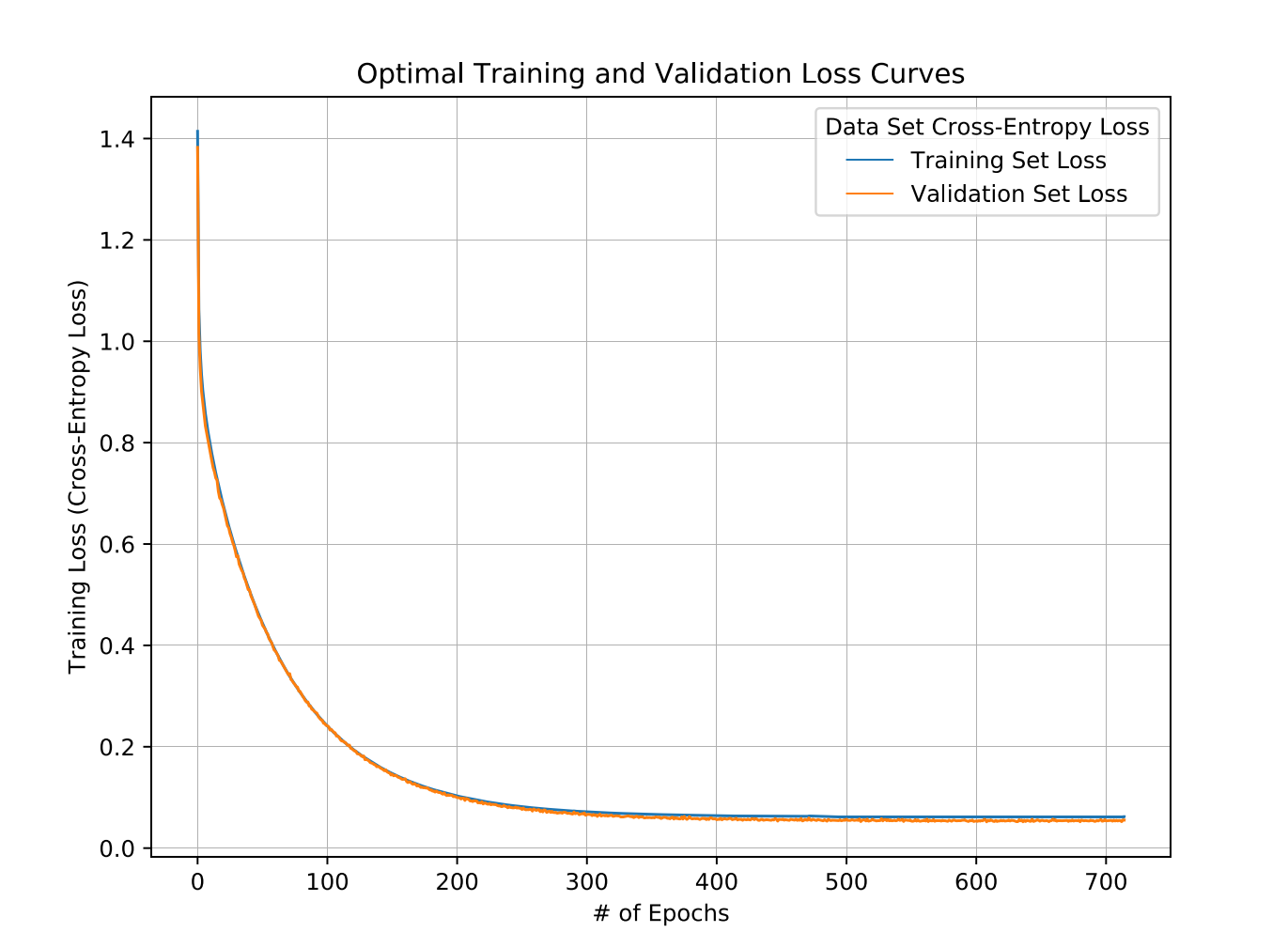
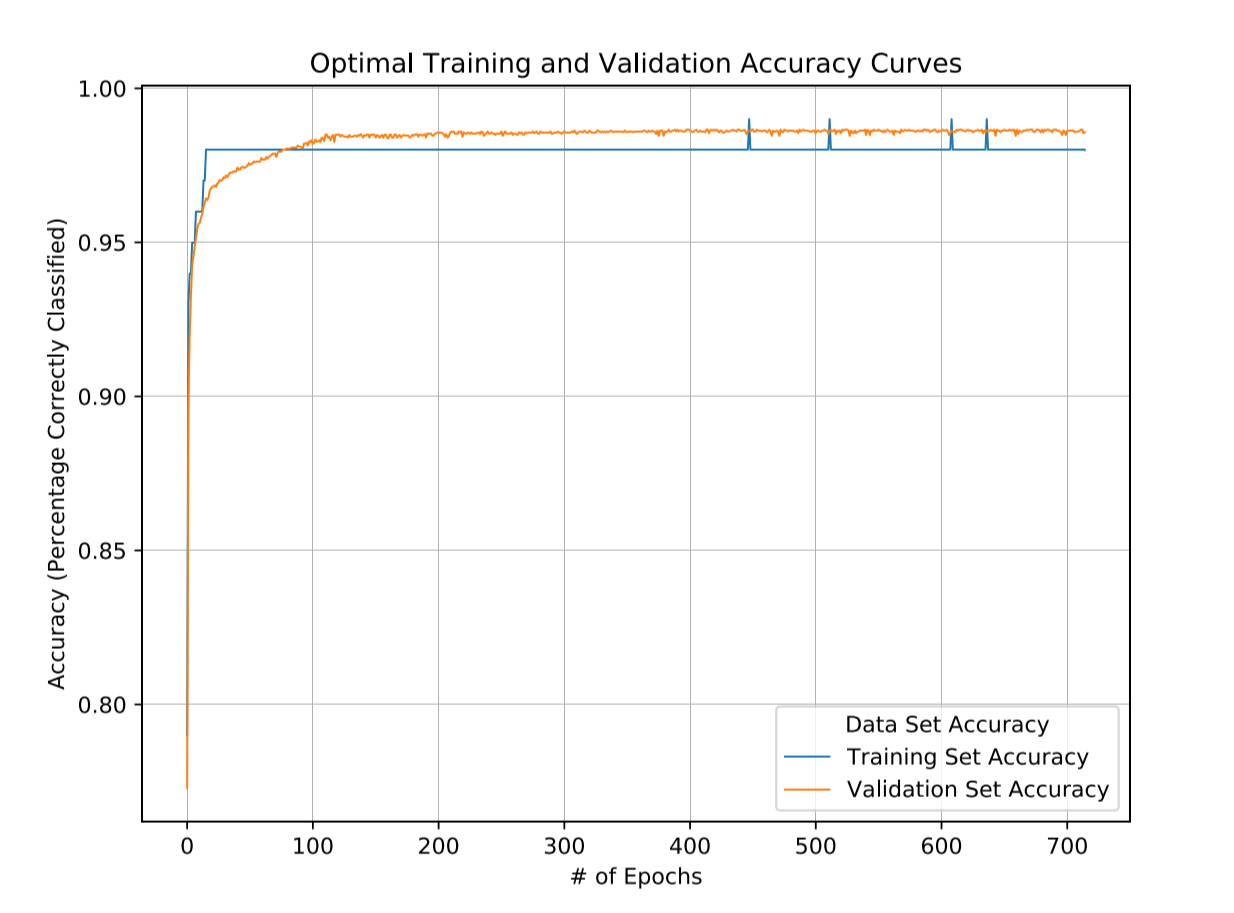
Where and , with all other constants defined as they were in Part 1 above. Code used to plot the following graphs can be found in *Appendix A: logistic\_regression.py*

## Part 2.1 – 1: Learning

Utilizing a decay coefficient and a mini-batch size with 5000 iterations, the following training curves were observed for different values of learning rates :



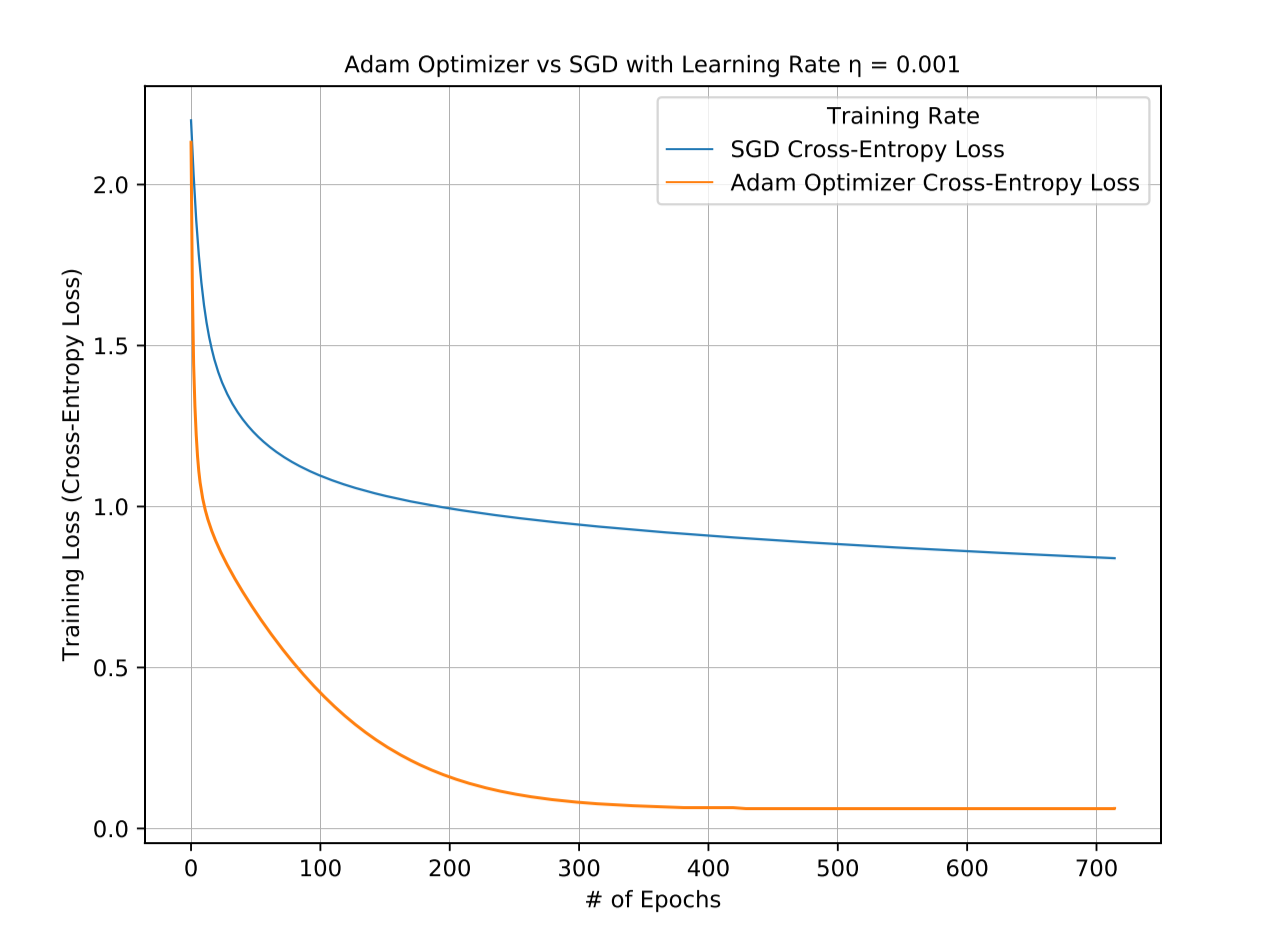
Selecting the optimal learning rate value of , the following training and validation curves for cross-entropy loss and classification accuracy were plotted:



From these curves, we observed a maximum test classification accuracy of 97.93%.

## Part 2.1 – 2: Beyond Plain SGD

Utilizing a decay coefficient , learning rate and a mini-batch size with 5000 iterations and the Adam Optimizer implementation provided by TensorFlow, we found the model to converge much faster than using the regular *GradientDescentOptimizer* used in the above sections. A plot comparing this convergence can be found below.



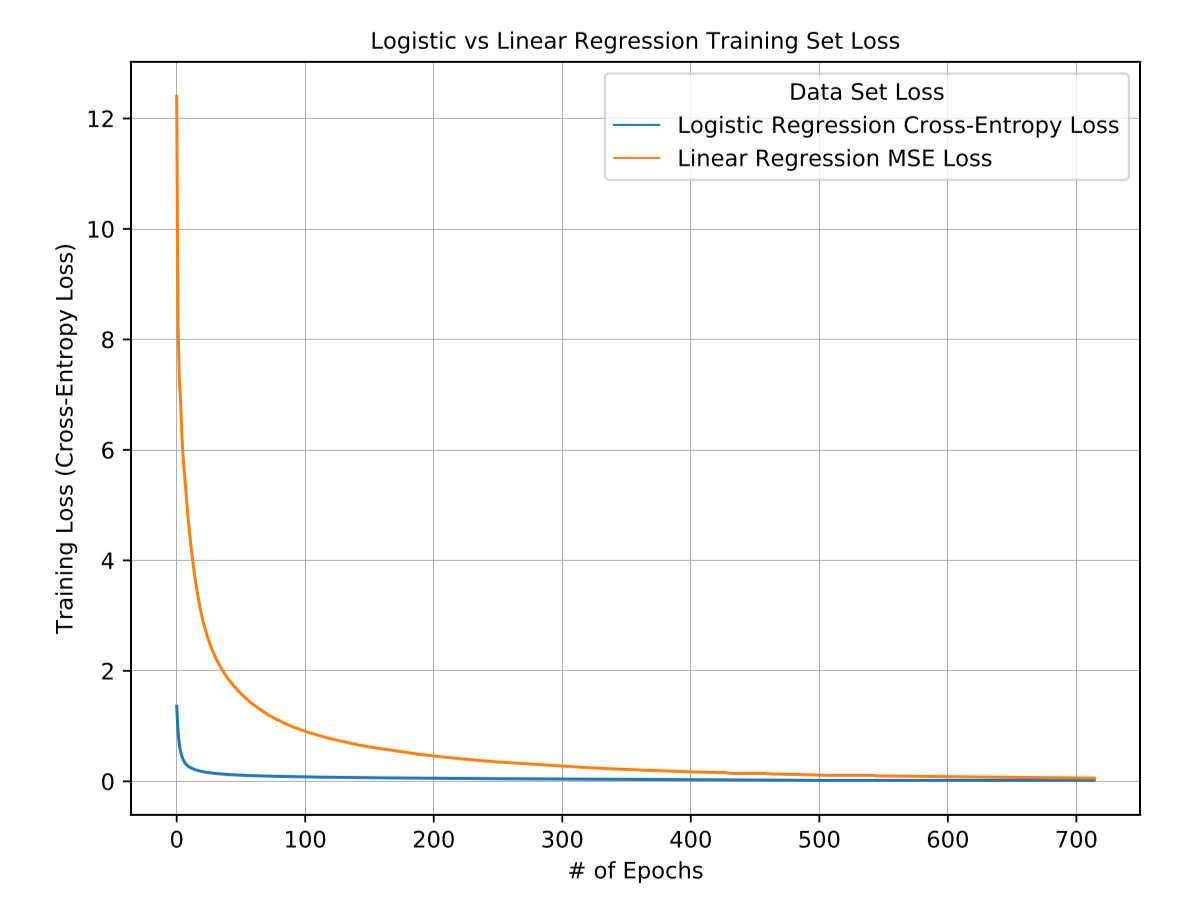
The Adam Optimizer can help to train on the notMNIST dataset with much fewer epochs while retaining the same or better level of loss and thereby predicted accuracy. This results in a faster model training time.

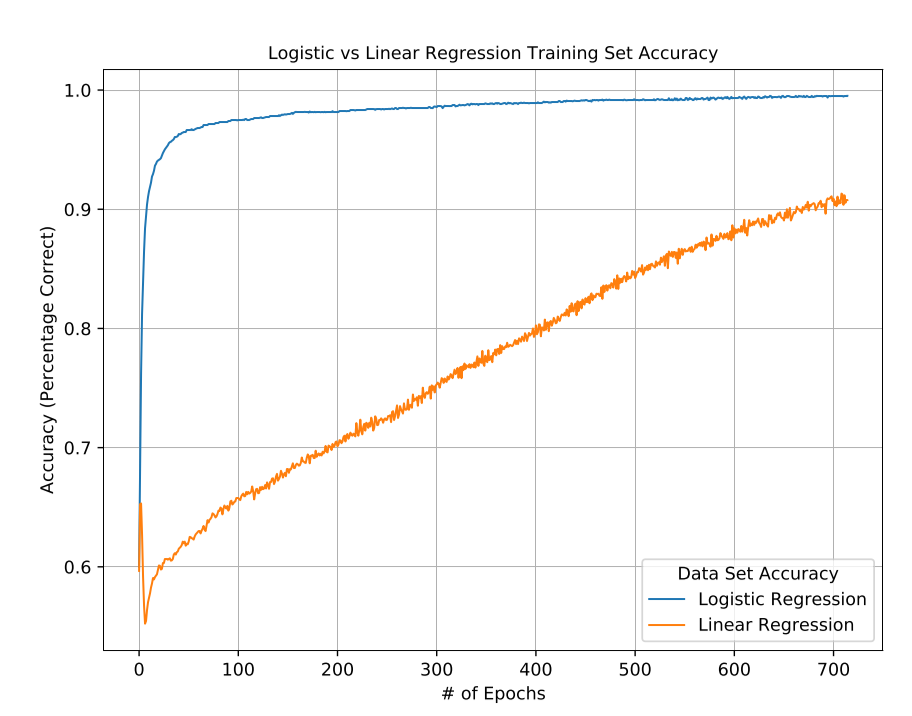
## Part 2.1 – 3: Comparison with linear regression

Setting , the optimal logistic regression learning rate was . Comparing the train, validation and test data set classification accuracy of this regression with the optimal linear regression classifier found using the “normal equation”, the following results were achieved:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Train Accuracy** | **Validation Accuracy** | **Test Accuracy** |
| **Adam Optimizer** | 0.99514 | 0.98 | 0.9793 |
| **Normal Equation** | 0.993 | 0.959 | 0.97 |

Furthermore, comparing the optimal logistic regression classifier with a learned linear regression classifier produced the following accuracy and loss curves for the training set:

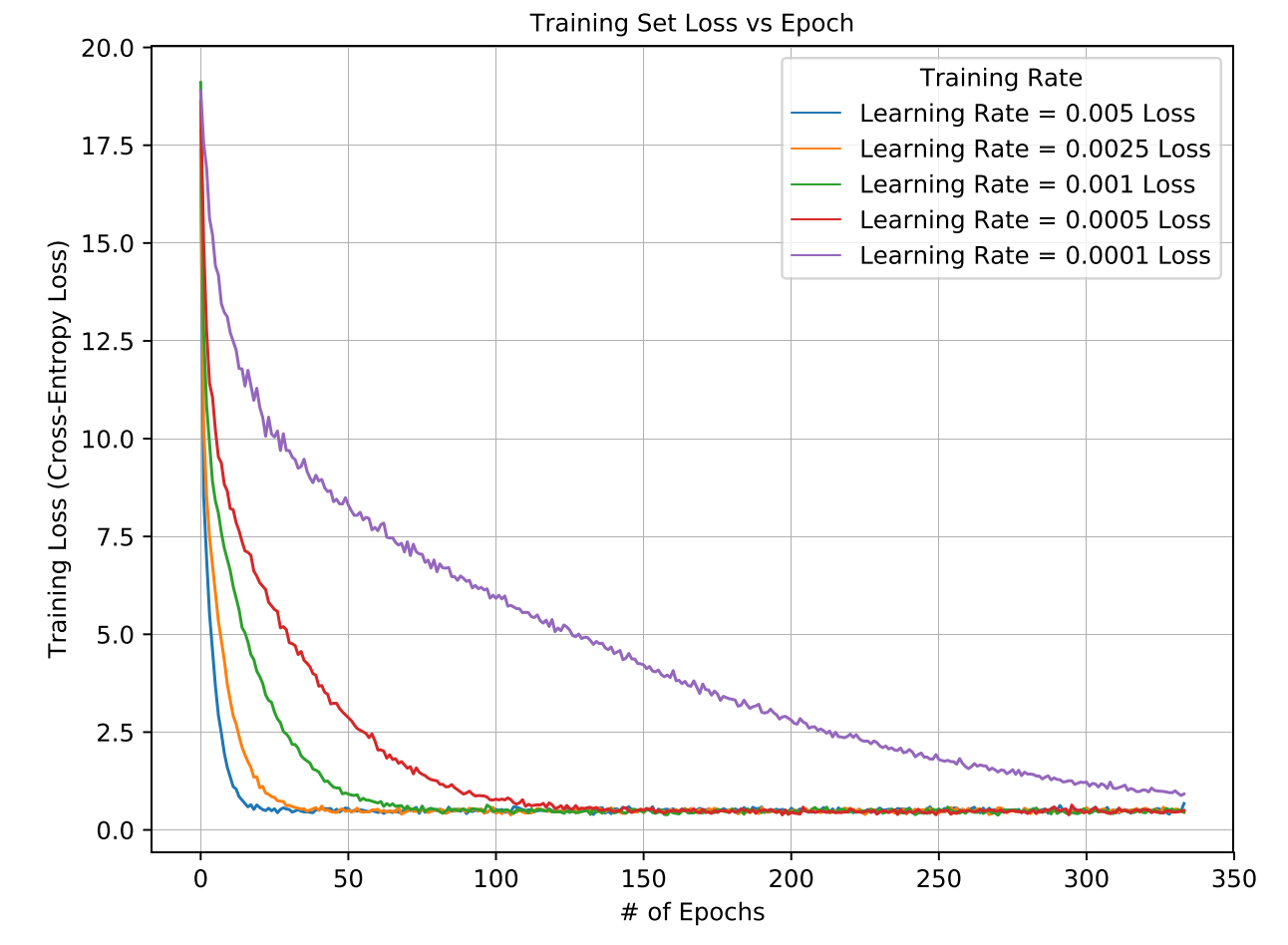


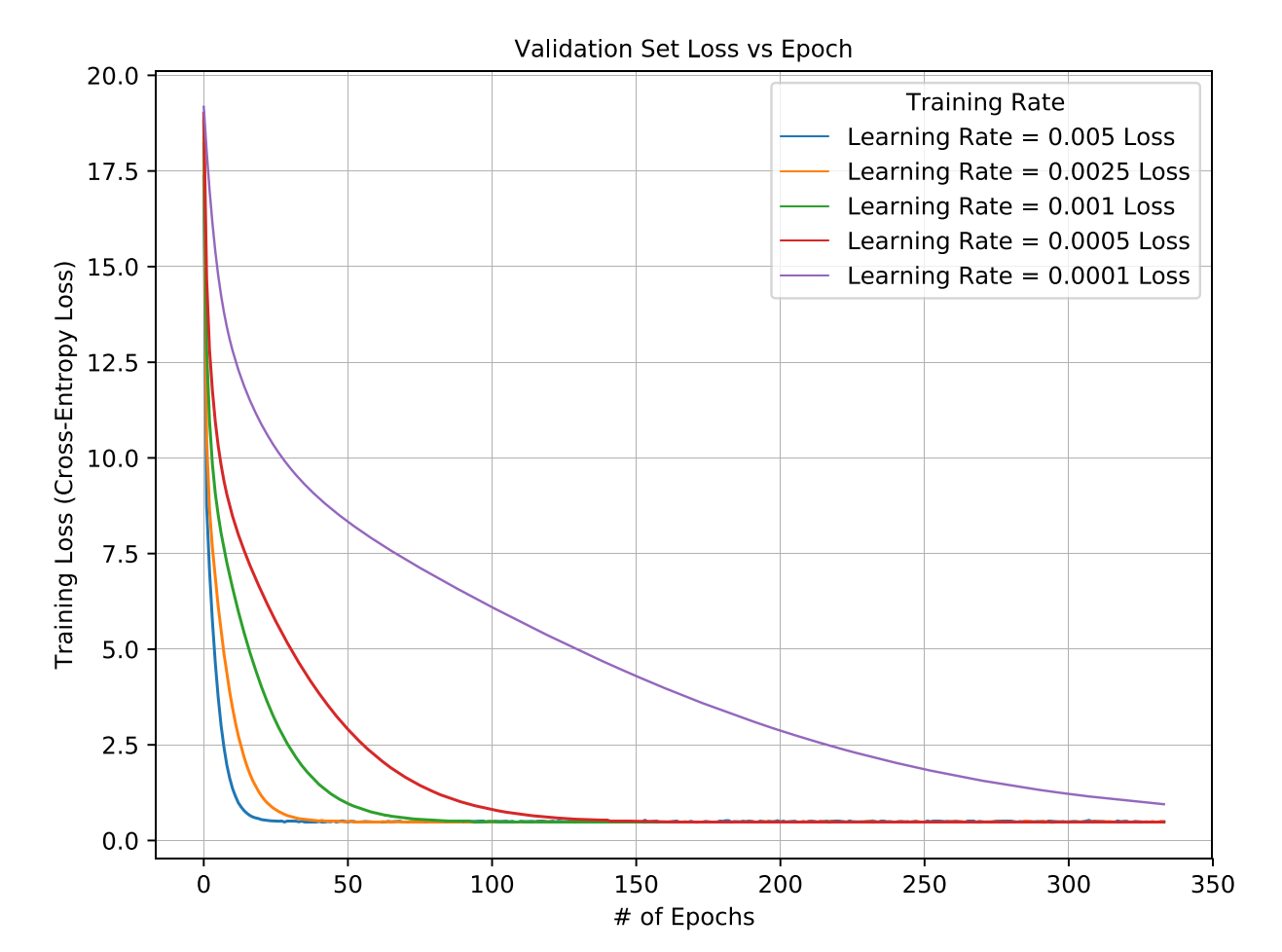


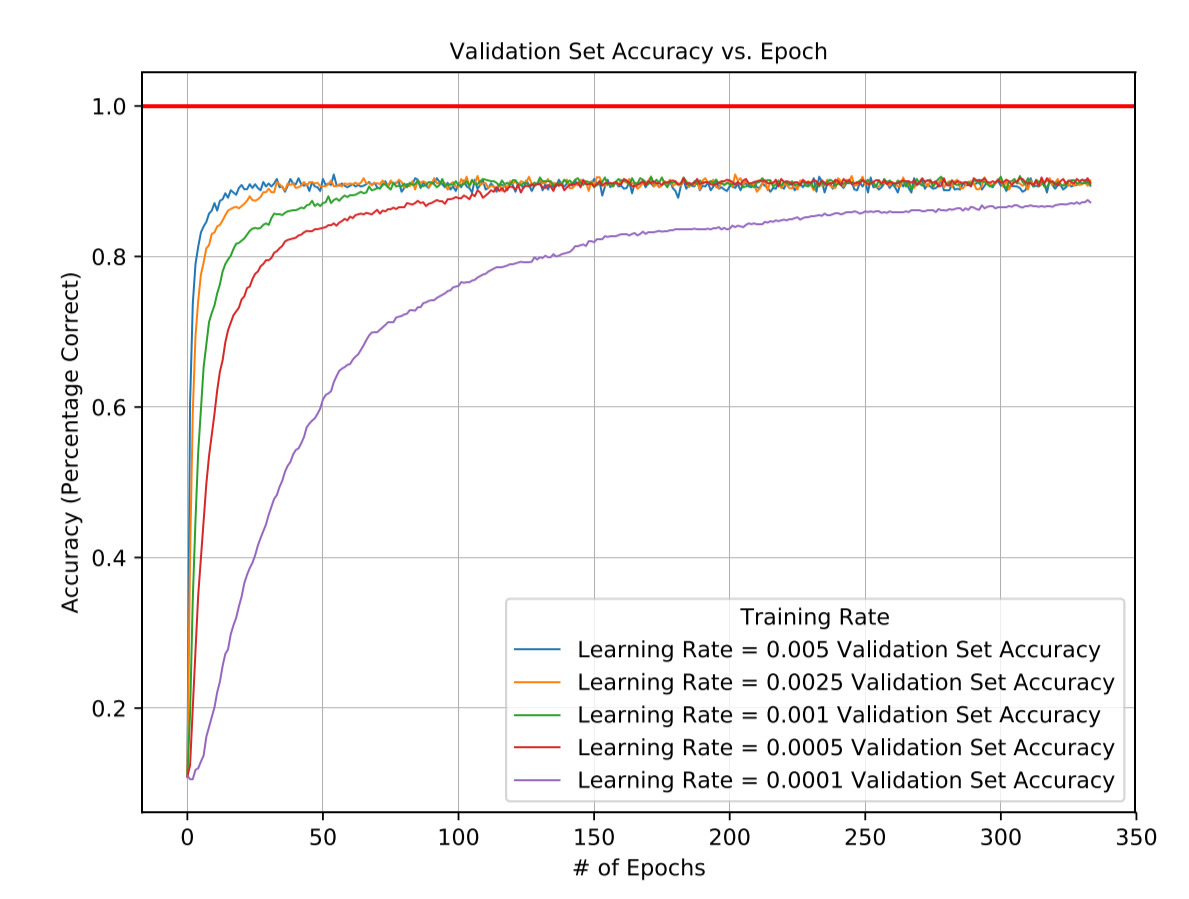
From these graphs, it is clear that Logistic regression converges much faster while also providing much greater accuracy over 730 epochs.

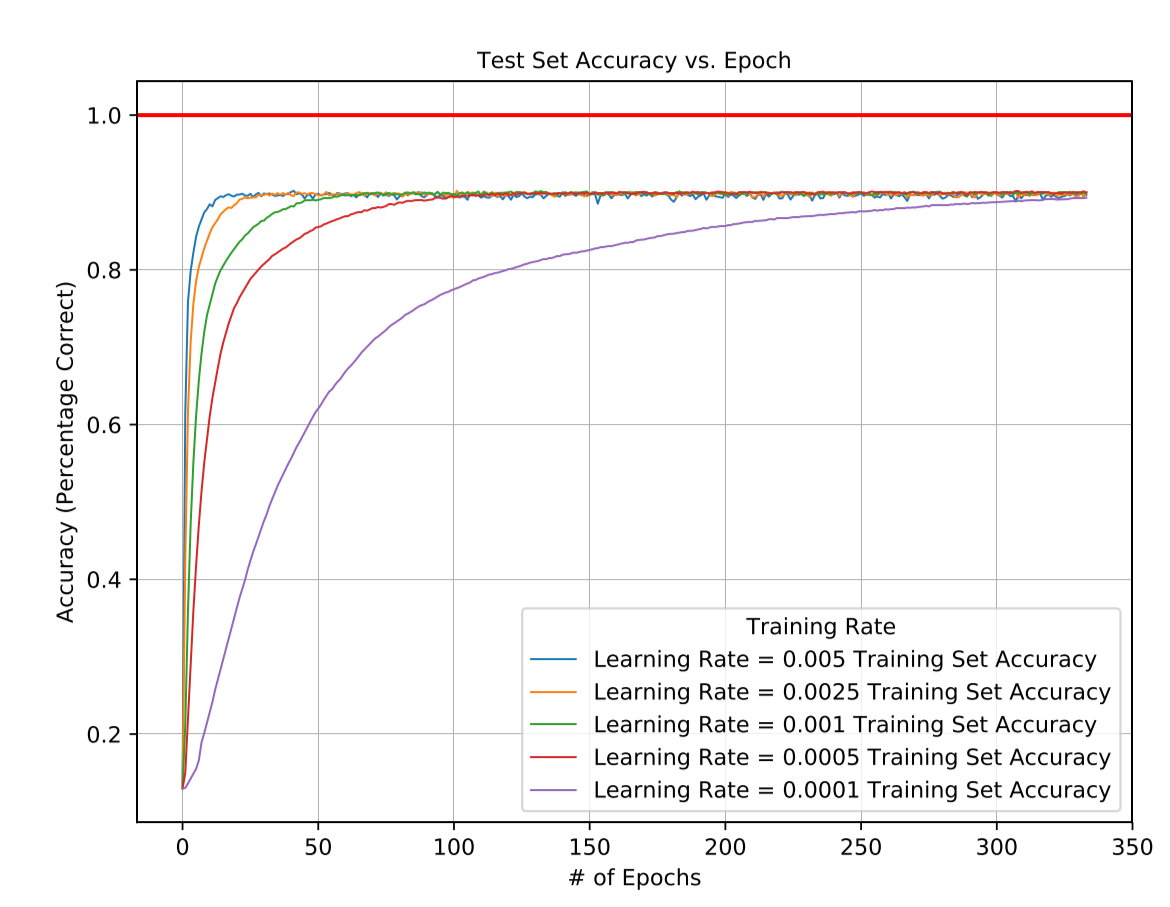
## Part 2.2 – 1: Multi-class Classification using Softmax Cross-Entropy on notMNIST

Utilizing similar code as in Part 2.1 with and , but substituting the loss function for the softmax cross-entropy function, the following training and validation curves for loss and accuracy were found:





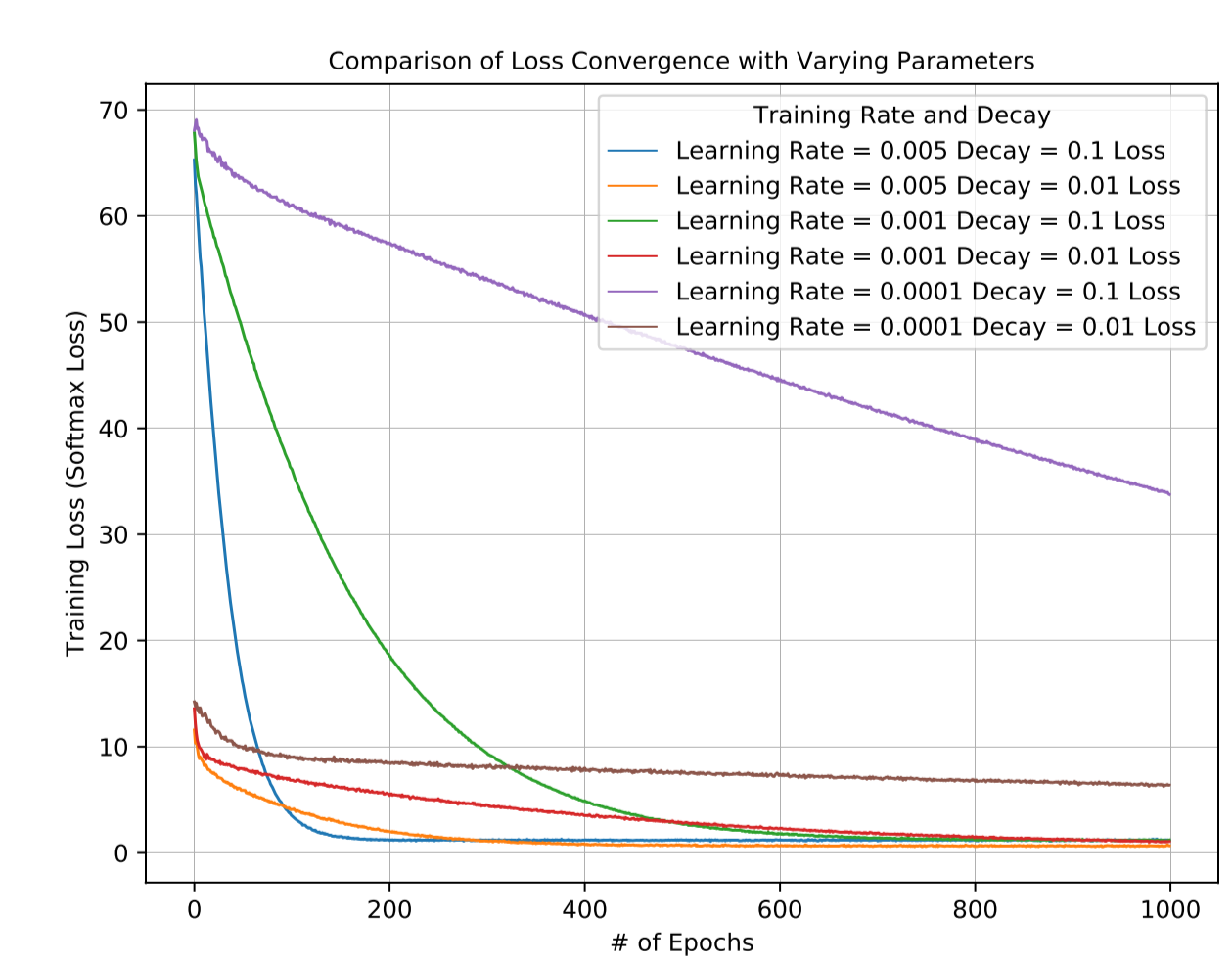




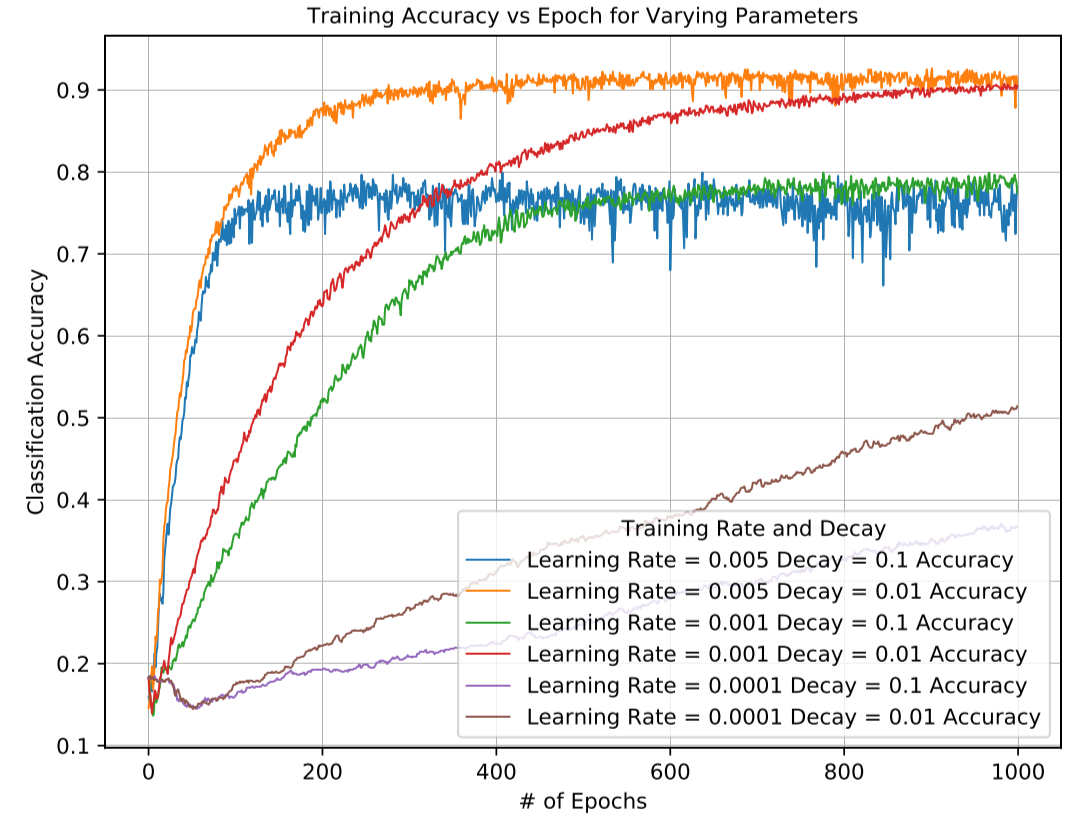
This lead to us selection as the optimal learning rate for this classifier. Using this learning rate, the best test classification accuracy was measured as 0.8961 or 89.61% correct. This is approximately 10% worse than in Part 2.1. Intuitively, this makes sense as there are far more classes than in the previous parts, making the model much more uncertain with its classifications.

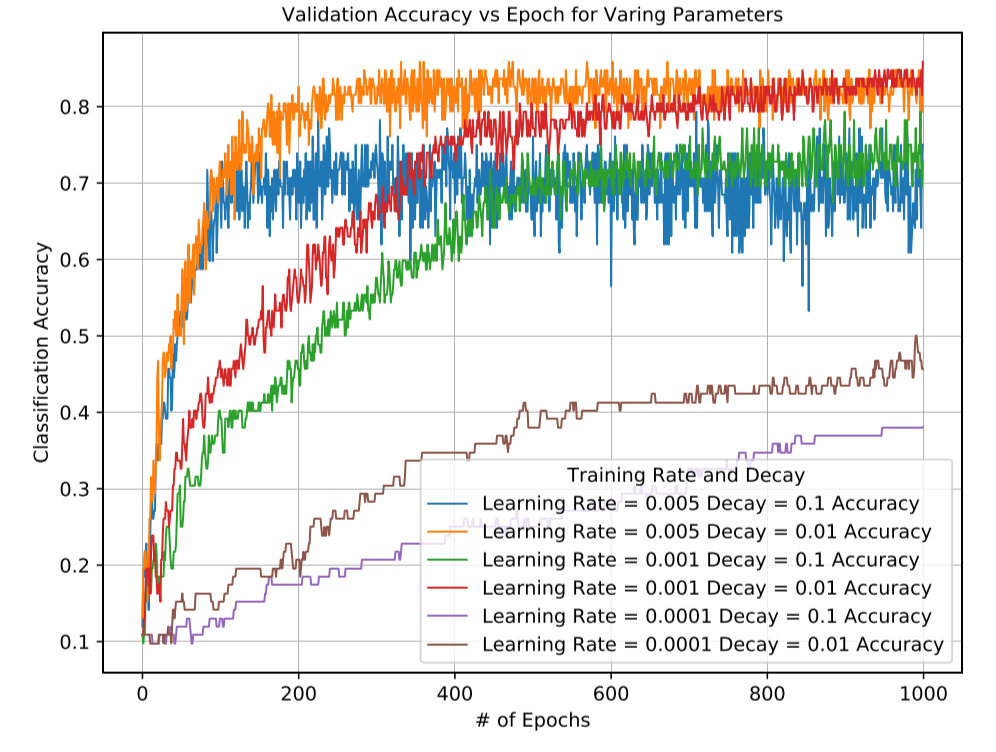
## Part 2.2 – 2: Multi-class Classification using Softmax Cross-Entropy on FaceScrub

Repeating the above part for the FaceScrub dataset with a batch size the following graphs were used to tune both the decay coefficient and learning rate :



Note that the above graph only covers a small sample of the parameters tested. For the full list, please refer to *Appendix A – logistic\_regression\_multi.py.* The training and validation accuracy and loss curves for these parameters is:





This resulted in optimal parameters and , with the best test classification accuracy being 89.25%. The best results achieved with the k-NN classifier in Assignment 1 for this dataset was 71.0%, meaning this logistic classifier was over 18 percentage points more accurate.

# Appendix A – Python code

## linear\_regression2.py

|  |
| --- |
| import \_pickle  import numpy as np  import tensorflow as tf  import matplotlib.pyplot as plt  import time  def load\_data():  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, trainTarget, validData, validTarget, testData, testTarget)  def SGD(xTrain, yTrain, batchSize, iters, learning\_rate, decay\_coefficient, use\_normal\_eqn = False):  """  Do stochastic gradient descent  :param xTrain: numpy array of shape (number of samples, width, height)  :param yTrain: np array of shape (number of samples, 1)  :param batchSize: integer  :param iters: integer  :param learning\_rate: float  :param decay\_coefficient: float  :param use\_normal\_eqn: bool  :return: losses for each epoch, list, each element is np.array of shape (1)  """  X = tf.placeholder(tf.float64, shape=(None, xTrain.shape[1]\*xTrain.shape[2]), name='X')  Y = tf.placeholder(tf.float64, shape=(None, 1), name='Y')  with tf.variable\_scope('weight', reuse=tf.AUTO\_REUSE):  weights = tf.get\_variable('weights', shape=[X.shape[1] + 1, 1], dtype=tf.float64) # (W\*H +1) x 1  x\_in = tf.pad(X, [[0, 0], [0, 1]], "CONSTANT", constant\_values=1) # bn x (W\*H +1)  z = tf.tensordot(x\_in, weights, axes=1)  dc = tf.placeholder(tf.float64, name='dc')  mse\_loss = tf.reduce\_mean(tf.square(Y-z))/(2)  decay\_loss = dc\*tf.sqrt(tf.reduce\_sum(tf.square(weights)))/2  total\_loss = mse\_loss + decay\_loss  full\_mse\_loss = tf.reduce\_mean(tf.square(Y-tf.tensordot(x\_in, weights, axes=1)))/2  full\_total\_loss = full\_mse\_loss + decay\_loss  if use\_normal\_eqn:  grads = [tf.reshape(tf.reduce\_mean(-tf.transpose((Y - z) \* x\_in) + tf.expand\_dims(dc,axis = 0) \* weights, axis=-1), (-1, 1)),]  else:  grads = tf.gradients(total\_loss, weights)  updates = tf.assign\_sub(weights, learning\_rate \* grads[0])  num\_sample = xTrain.shape[0]  num\_epochs = (batchSize\*iters + 1)//num\_sample + 1  curr\_iter = 0  losses = []  with tf.Session() as sess:  sess.run(tf.global\_variables\_initializer())  for i in range(num\_epochs):  inds = np.arange(num\_sample)  np.random.shuffle(inds)  this\_epoch\_loss = 0.0  curr\_samples = 0  for j in range((num\_sample+1)//batchSize + 1):  l = j\*batchSize  u = min(xTrain.shape[0], l+batchSize)  if l == u:  continue  if curr\_iter >= iters:  break  curr\_iter += 1  curr\_samples += (u - l)  this\_inds = inds[l:u]  this\_batch\_X = xTrain[this\_inds]  this\_batch\_Y = yTrain[this\_inds]  this\_batch\_X = np.reshape(this\_batch\_X, (this\_batch\_X.shape[0], this\_batch\_X.shape[1]\*this\_batch\_X.shape[2]))  this\_loss, gr, wts, full\_loss = sess.run([total\_loss, grads, weights, full\_total\_loss],  feed\_dict={X: this\_batch\_X, Y:this\_batch\_Y.astype(np.float64),  dc:np.array([decay\_coefficient], dtype=np.float64)})  this\_epoch\_loss += this\_loss\*(u-l)  sess.run(updates,  feed\_dict={X: this\_batch\_X, Y:this\_batch\_Y.astype(np.float64),  dc:np.array([decay\_coefficient], dtype=np.float64)})  if curr\_samples == 0:  continue  this\_epoch\_loss /= curr\_samples  losses.append(full\_loss)  return [losses, wts]  def tuning\_the\_learning\_rate():  iters = 20000  batch\_size = 500  learning\_rates = [0.005, 0.001, 0.0001]  decay\_coefficient = 0.  (trainData, trainTarget,  testData, testTarget,  validData, validTarget) = load\_data()  plt.figure(figsize=(8, 6), dpi=80)  plt.subplot(1, 1, 1)  for i in range(0,len(learning\_rates)):  loss, \_ = SGD(trainData, trainTarget, batch\_size, iters, learning\_rates[i], decay\_coefficient)  x = np.arange(len(loss))  plt.plot(x, np.array(loss),label = r'$\eta$' + " = " + str(learning\_rates[i]))  plt.xlim(-200, 3000)  plt.xlabel('Number of Epoch')  plt.ylabel('MSE')  plt.title("MSE vs Epoch")  plt.legend()  plt.show()    plt.savefig("Tuning the Learning Rate.pdf", format="pdf")    return plt  def effect\_of\_minibatch\_size():  iters = 20000  batch\_sizes = [500, 1500, 3500]  learning\_rate = 0.005  decay\_coefficient = 0  losses = []    (trainData, trainTarget,  testData, testTarget,  validData, validTarget) = load\_data()    for i in range(0,len(batch\_sizes)):  time\_start = time.clock()  loss = SGD(trainData, trainTarget, batch\_sizes[i], iters, learning\_rate, decay\_coefficient)[0]  x = np.arange(len(loss))  plt.plot(x, np.array(loss), label='batch\_size = %f' % batch\_sizes[i])  losses.append(loss)  time\_elapsed = (time.clock() - time\_start)  print("Time passed for B = " + str(batch\_sizes[i]) + ": " + str(time\_elapsed))    plt.xlabel('Number of Epoch')  plt.ylabel('MSE')  plt.title("MSE vs Epoch")  plt.legend()  plt.show()  return losses  def generalization():  iters = 20000  batch\_size = 500  learning\_rate = 0.005  decay\_coefficients = [0.0, 0.001, 0.1, 1]  (trainData, trainTarget,  testData, testTarget,  validData, validTarget) = load\_data()  final\_weights = []  validation\_accuracy = []  test\_accuracy = [];  for i in range(0,len(decay\_coefficients)):  time\_start = time.clock()  (loss, wt) = SGD(trainData, trainTarget, batch\_size, iters, learning\_rate, decay\_coefficients[i])  time\_elapsed = (time.clock() - time\_start)  print("Time passed for dc = " + str(decay\_coefficients[i]) + ": " + str(time\_elapsed))  x = np.arange(len(loss))  plt.plot(x, np.array(loss), label='decay\_coefficient = %f' % decay\_coefficients[i])  final\_weights.append(wt)    valid\_linear = np.reshape(validData, (validData.shape[0], validData.shape[1]\*validData.shape[2]))  x\_in\_valid = tf.pad(valid\_linear, [[0, 0], [0, 1]], "CONSTANT", constant\_values=1)  valid\_y\_pred = tf.matmul(x\_in\_valid, wt)  same\_valid = tf.equal(tf.greater(valid\_y\_pred, tf.constant(0.5, tf.float64)), tf.constant(validTarget, tf.bool))  v\_accuracy = tf.count\_nonzero(same\_valid) / tf.constant(validTarget).shape[0]  with tf.Session() as sess:  validation\_accuracy.append(sess.run(v\_accuracy))    test\_linear = np.reshape(testData, (testData.shape[0], testData.shape[1]\*testData.shape[2]))  x\_in\_test = tf.pad(test\_linear, [[0, 0], [0, 1]], "CONSTANT", constant\_values=1)  test\_y\_pred = tf.matmul(x\_in\_test, wt)  same\_test = tf.equal(tf.greater(test\_y\_pred, tf.constant(0.5, tf.float64)), tf.constant(testTarget, tf.bool))  t\_accuracy = tf.count\_nonzero(same\_test) / tf.constant(testTarget).shape[0]  with tf.Session() as sess:  test\_accuracy.append(sess.run(t\_accuracy))    plt.xlabel('Number of Epoch')  plt.ylabel('MSE')  plt.title("MSE vs Epoch")  plt.legend()  plt.show()    return (validation\_accuracy, test\_accuracy)  def sgd\_vs\_normal\_equation():  iters = 20000  batch\_size = 500  learning\_rate = 0.005  decay\_coefficient = 0  (trainData, trainTarget,  testData, testTarget,  validData, validTarget) = load\_data()  validation\_accuracy = []  test\_accuracy = []    time\_start = time.clock()  (loss, wt) = SGD(trainData, trainTarget, batch\_size, iters, learning\_rate, decay\_coefficient)  time\_elapsed = (time.clock() - time\_start)  print("Time passed for SGD = : " + str(time\_elapsed))    valid\_linear = np.reshape(validData, (validData.shape[0], validData.shape[1]\*validData.shape[2]))  x\_in\_valid = tf.pad(valid\_linear, [[0, 0], [0, 1]], "CONSTANT", constant\_values=1)  valid\_y\_pred = tf.matmul(x\_in\_valid, wt)  same\_valid = tf.equal(tf.greater(valid\_y\_pred, tf.constant(0.5, tf.float64)), tf.constant(validTarget, tf.bool))  v\_accuracy = tf.count\_nonzero(same\_valid) / tf.constant(validTarget).shape[0]  with tf.Session() as sess:  validation\_accuracy.append(sess.run(v\_accuracy))    test\_linear = np.reshape(testData, (testData.shape[0], testData.shape[1]\*testData.shape[2]))  x\_in\_test = tf.pad(test\_linear, [[0, 0], [0, 1]], "CONSTANT", constant\_values=1)  test\_y\_pred = tf.matmul(x\_in\_test, wt)  same\_test = tf.equal(tf.greater(test\_y\_pred, tf.constant(0.5, tf.float64)), tf.constant(testTarget, tf.bool))  t\_accuracy = tf.count\_nonzero(same\_test) / tf.constant(testTarget).shape[0]  with tf.Session() as sess:  test\_accuracy.append(sess.run(t\_accuracy))    time\_start = time.clock()  train\_linear = np.reshape(trainData, (trainData.shape[0], trainData.shape[1]\*trainData.shape[2]))  x\_in\_train = tf.pad(train\_linear, [[0, 0], [0, 1]], "CONSTANT", constant\_values=1)  y\_in\_train = tf.constant(trainTarget, tf.float64)  inv = tf.matrix\_inverse(tf.matmul(x\_in\_train,x\_in\_train,True))  wt = tf.matmul(inv, tf.matmul(x\_in\_train,y\_in\_train, True))  time\_elapsed = (time.clock() - time\_start)  print("Time passed for normal = : " + str(time\_elapsed))    z = tf.matmul(wt, x\_in\_train, True, True)  n\_loss = tf.reduce\_mean(tf.square(y\_in\_train-tf.transpose(z)))/(2)  with tf.Session() as sess:  normal\_loss = sess.run(n\_loss)    train\_linear = np.reshape(trainData, (trainData.shape[0], trainData.shape[1]\*trainData.shape[2]))  x\_in\_train = tf.pad(train\_linear, [[0, 0], [0, 1]], "CONSTANT", constant\_values=1)  train\_y\_pred = tf.matmul(x\_in\_train, wt)  same\_train = tf.equal(tf.greater(train\_y\_pred, tf.constant(0.5, tf.float64)), tf.constant(trainTarget, tf.bool))  tr\_accuracy = tf.count\_nonzero(same\_train) / tf.constant(trainTarget).shape[0]  with tf.Session() as sess:  print(sess.run(tr\_accuracy))    valid\_linear = np.reshape(validData, (validData.shape[0], validData.shape[1]\*validData.shape[2]))  x\_in\_valid = tf.pad(valid\_linear, [[0, 0], [0, 1]], "CONSTANT", constant\_values=1)  valid\_y\_pred = tf.matmul(x\_in\_valid, wt)  same\_valid = tf.equal(tf.greater(valid\_y\_pred, tf.constant(0.5, tf.float64)), tf.constant(validTarget, tf.bool))  v\_accuracy = tf.count\_nonzero(same\_valid) / tf.constant(validTarget).shape[0]  with tf.Session() as sess:  validation\_accuracy.append(sess.run(v\_accuracy))    test\_linear = np.reshape(testData, (testData.shape[0], testData.shape[1]\*testData.shape[2]))  x\_in\_test = tf.pad(test\_linear, [[0, 0], [0, 1]], "CONSTANT", constant\_values=1)  test\_y\_pred = tf.matmul(x\_in\_test, wt)  same\_test = tf.equal(tf.greater(test\_y\_pred, tf.constant(0.5, tf.float64)), tf.constant(testTarget, tf.bool))  t\_accuracy = tf.count\_nonzero(same\_test) / tf.constant(testTarget).shape[0]  with tf.Session() as sess:  test\_accuracy.append(sess.run(t\_accuracy))    x = np.arange(len(normal\_loss))  plt.plot(x, np.array(loss), label='default\_gradient')  plt.plot(x, np.array(normal\_loss), label='normal\_gradient')  plt.xlabel('Number of Epoch')  plt.ylabel('MSE')  plt.title("MSE vs Epoch")  plt.legend()  plt.show()      return (loss, normal\_loss, validation\_accuracy, test\_accuracy)  if \_\_name\_\_ == '\_\_main\_\_':  tuning\_the\_learning\_rate()  losses = effect\_of\_minibatch\_size()  (valid, test) = generalization()  (l, nl, valid2, test2) = sgd\_vs\_normal\_equation() |