Assignment 1

Wednesday, February 6th, 2019 | ECE1513 | Dylan Johnston | Omar Ismail
University of Toronto | Introduction to Machine Learning | 1003852690 | 999467660

Please note that both partners contributed equally to the assignment, 50% contribution each to the final result.

1 Linear Regression

1.1 Loss Function and Gradient

Linear regression uses a least squares loss function of the form:

$$\mathcal{F}_{MSE} = \sum_{i=1}^{N} \frac{(y_i - \hat{y}_i)^2}{2N} + \mathcal{F}_{reg}(\lambda, \underline{w}) = \sum_{i=1}^{N} \frac{1}{2N}||y_i - [\underline{x}_i^T \underline{w} + b]||^2 + \frac{\lambda}{2}||\underline{w}||^2 \tag{1}$$

Where our prediction for any data point, $\hat{y}_i$, is simply the data point multiplied the current weight vector, with a bias added. When the prediction for a data point is correct, the sum is zero and that data point will not contribute to the error function. The loss function will then return a sum of all the predictions that were incorrect within the data set. Note that a regularization term has been added to the cost function, which is used to penalize the complexity of our solution, in order to prevent over-fitting of the training data. The vectorized form of the MSE loss function is given by:

$$\mathcal{F}_{MSE} = \frac{(\underline{y} - (\mathbf{x}^T\underline{w} + \underline{b}))^T(\underline{y} - (\mathbf{x}^T\underline{w} + \underline{b}))}{2N} + \frac{\lambda \underline{w}^T \underline{w}}{2} \tag{2}$$

Where both $\underline{b}$ and $\underline{w}^T\underline{w}$ are column vectors that have the same length as $\underline{y}$ with the value of b or $||w||_2^2$ respectively repeated. Mean square error is used so that we do not need to consider whether the point is a negative or positive displacement from the regression line, since minimizing the square distances is equivalent to minimizing the distances themselves. We will use a gradient descent method to update the weights and bias for each iteration. This involves taking the derivative of the loss function with respect to each variable:

$$\underline{w}_{j+1} = \underline{w}_j - \alpha \frac{\partial \mathcal{F}_{MSE}}{\partial \underline{w}} \quad , \quad b_{j+1} = b_j - \alpha \frac{\partial \mathcal{F}_{MSE}}{\partial b} \tag{3}$$

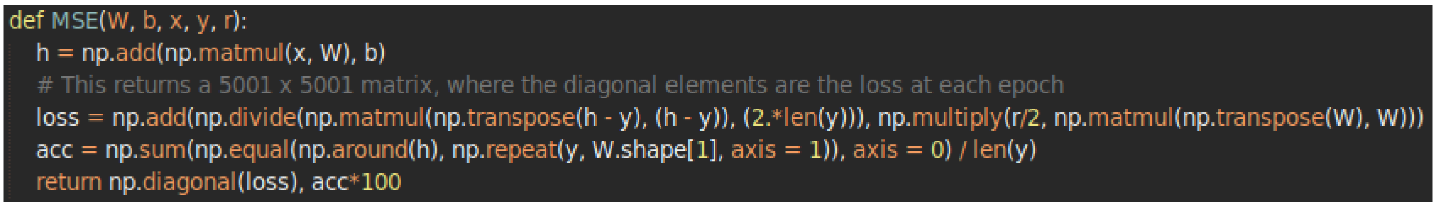
Where α is our learning rate, one of the hyperparameters of this optimization. We will initialize our weights and bias with random values, and iterate to convergence. These derivatives are given below:

$$\frac{\partial \mathcal{F}_{MSE}}{\partial b} = \sum_{i=1}^{N} \frac{1}{N}([\underline{x}_i^T \underline{w} + b] - y_i) \quad , \quad \frac{\partial \mathcal{F}_{MSE}}{\partial \underline{w}} = \sum_{i=1}^{N} \frac{1}{N}\Big\{([\underline{x}_i^T \underline{w} + b] - y_i)\underline{x}_i^T\Big\} + \lambda||\underline{w}||$$

The vectorized form of these derivatives is also provided below.

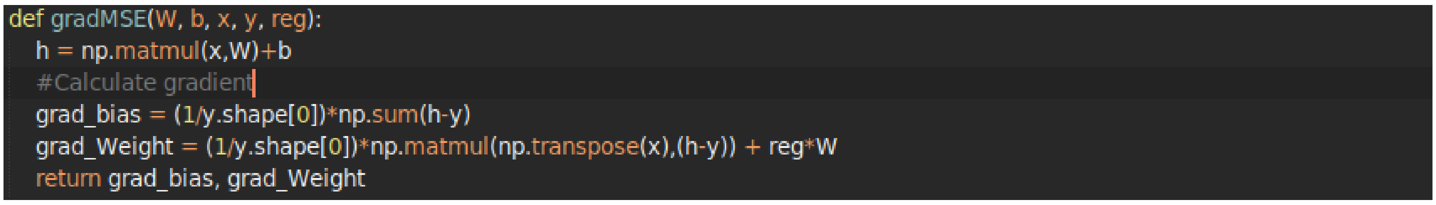
$$\frac{\partial \mathcal{F}_{MSE}}{\partial b} = \frac{\{\boldsymbol{x}^T\underline{w} + \underline{b} - \underline{y}\}}{N} \quad , \quad \frac{\partial \mathcal{F}_{MSE}}{\partial \underline{w}} = \frac{\boldsymbol{x}^T\{\boldsymbol{x}^T\underline{w} + \underline{b} - \underline{y}\}}{N} + \lambda\underline{w} \tag{4}$$

The code snippets for both the MSE cost function and its gradients are provided below:



```
def MSE(W, b, x, y, r):
    h = np.add(np.matmul(x, W), b)
    # This returns a 5001 x 5001 matrix, where the diagonal elements are the loss at each epoch
    loss = np.add(np.divide(np.matmul(np.transpose(h - y), (h - y)), (2.*len(y))), np.multiply(r/2, np.matmul(np.transpose(W), W)))
    acc = np.sum(np.equal(np.around(h), np.repeat(y, W.shape[1], axis = 1)), axis = 0) / len(y)
    return np.diagonal(loss), acc*100
```

Figure 1: Python code snippet of the implemented vectorized MSE cost function.



```
def gradMSE(W, b, x, y, reg):
    h = np.matmul(x,W)+b
    #Calculate gradient
    grad_bias = (1/y.shape[0])*np.sum(h-y)
    grad_Weight = (1/y.shape[0])*np.matmul(np.transpose(x),(h-y)) + reg*W
    return grad_bias, grad_Weight
```

Figure 2: Python code snippet of the implemented vectorized MSE cost function gradients.

1.2 Gradient Descent Implementation

Now that we have defined methods for calculating the gradient of the cost function for both the model weights and biases, they will be used to iterate the value for the model weights and biases towards an optimized value. For each step in our iteration (each epoch), the following algorithm will be followed:

- Use gradient expressions to determine derivative of cost function with respect to weights and biases
- Update the new value for the weights and biases using equations 3 and iterate until convergence
- Return trained weights and biases, which can be used once in our mean square error loss function to return a vector of the total loss for each training epoch, as well as the accuracy, for the training, validation and test data sets.

1.3 Tuning the Learning Rate

Figures 3 and 4 show the calculated loss and accuracy of the linear regression model for of 5000 epochs. As stated in the question, the regularization parameter λ is set to zero, and the learning rate α is varied between the values $\alpha = \{0.005, 0.001, 0.0001\}$. The data plotted in blue corresponds to the training data, orange to the validation data, and green to the test data.

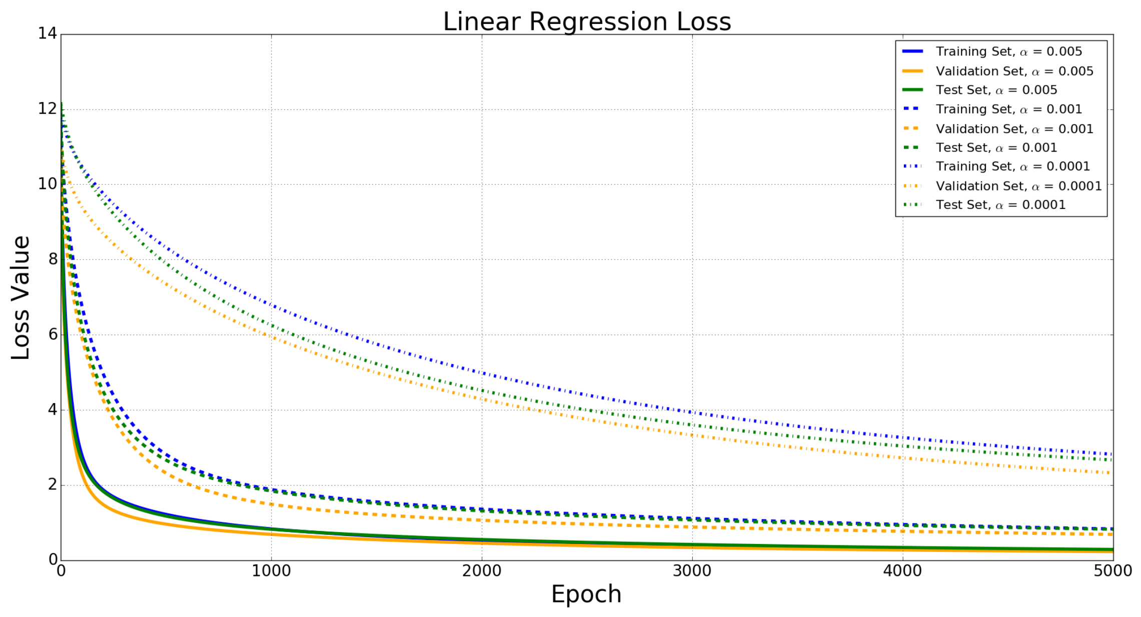


Figure 3: Calculated MSE loss at each epoch of 5000 training epochs, with regularization $\lambda = 0$, for learning rate $\alpha = \{0.005, 0.001, 0.0001\}$.

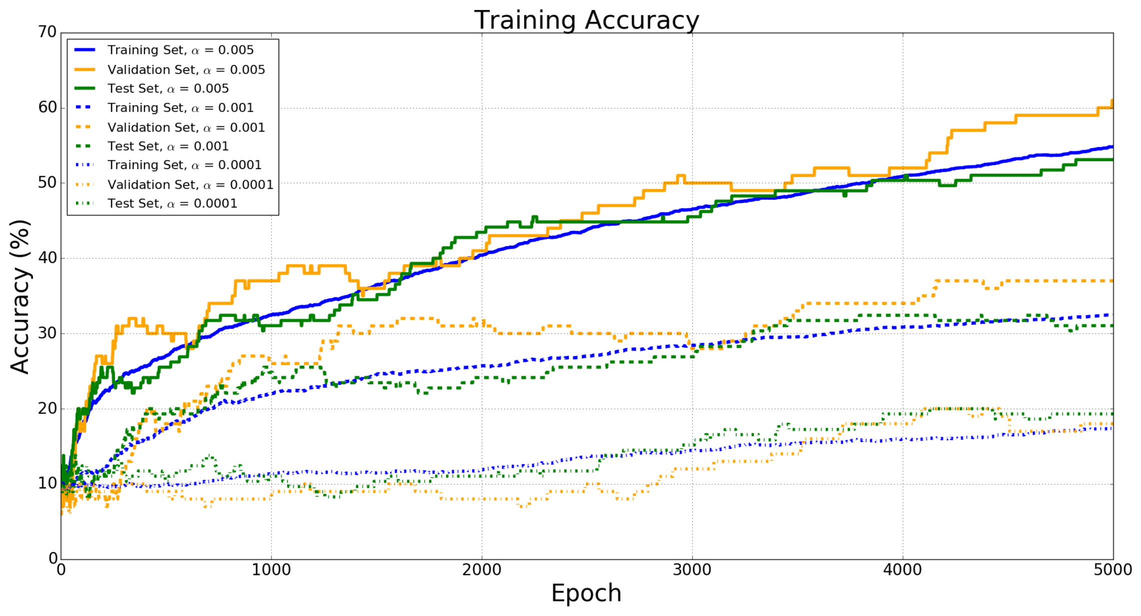


Figure 4: Calculated accuracy at each epoch of 5000 training epochs, with regularization $\lambda = 0$, for learning rate $\alpha = \{0.005, 0.001, 0.0001\}$.

When running this segment of code, the time required to train the weights was recorded. It seems that the learning rate α seems to have little impact on the training time of the weights and biases. The training time was consistently between 14 and 20 seconds, and there was no obvious trend in training time reduction by varying the learning rate. This makes sense, because the same steps are being performed, for the same number of iterations, with just a slightly different constant value.

As per Figures 3 and 4, the final accuracy approached 80% with the largest value of learning rate, while for the smallest learning rate, the final accuracy only approached around 20%. Clearly, smaller learning rates will require more iterations to reach the same final classification accuracy as a larger learning rate. Thus, the learning rate hyperparameter is an important choice when trying to optimize a machine learning algorithm.

1.4 Generalization

Figures 5 and 6 show the calculated loss and accuracy of the linear regression model for of 5000 epochs. As stated in the question, the learning rate α is set to 0.005, and the regularization parameter λ is varied between the values $\lambda = \{0.001, 0.1, 0.5\}$. The data plotted in blue corresponds to the training data, orange to the validation data, and green to the test data.

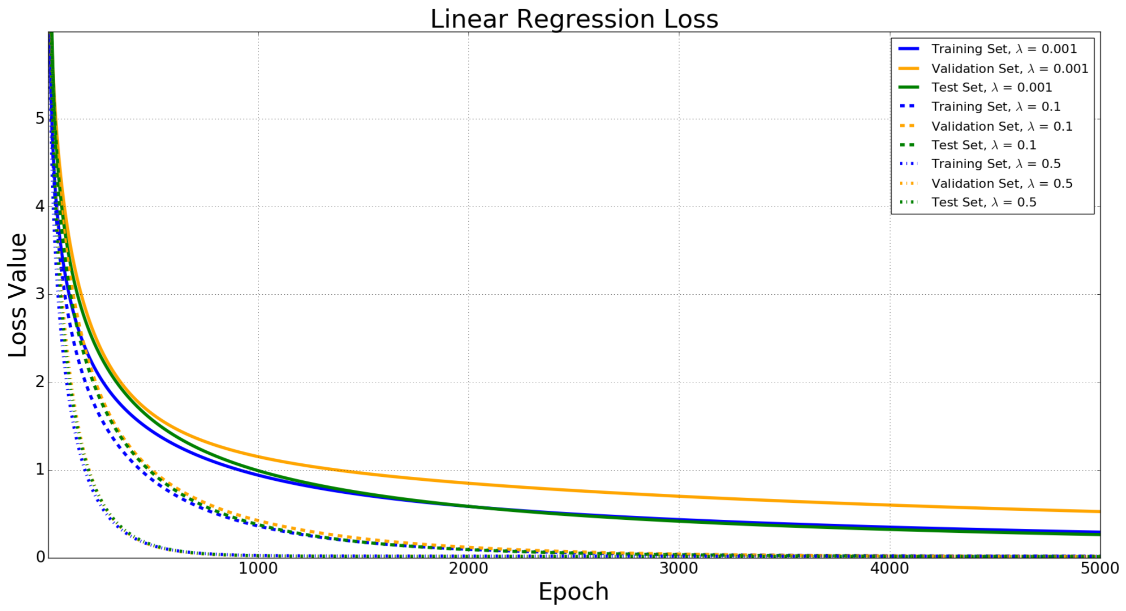


Figure 5: Calculated MSE loss at each epoch of 5000 training epochs, with learning rate $\alpha = 0.005$, for regularization $\lambda = \{0.001, 0.1, 0.5\}$.

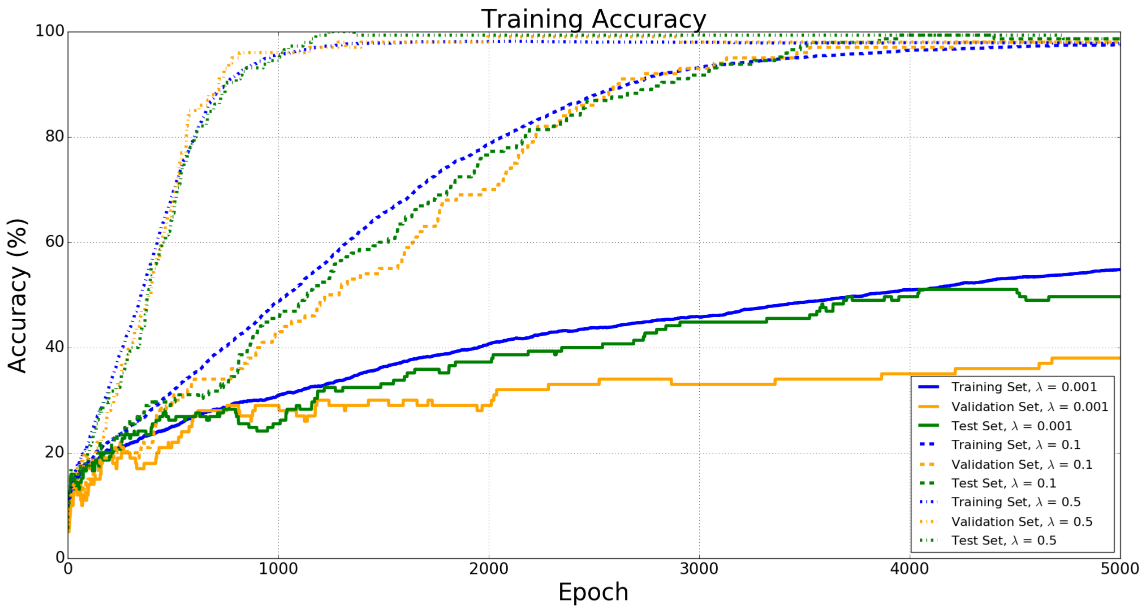


Figure 6: Calculated accuracy at each epoch of 5000 training epochs, with learning rate $\alpha = 0.005$, for regularization $\lambda = \{0.001, 0.1, 0.5\}$.

Based on the Figures from question 1.3, as well as Figures 5 and 6, including a small regularization parameter, namely $\lambda = 0.001$, seems to have little effect on the resulting optimized model, since the loss and accuracy curve are nearly identical to when there was no regularization, and $\alpha = 0.005$. When the value of this regularization is increased, however, it seems to cause the model to not only optimize quicker, but to also achieve a better final classification accuracy, reaching approximately 95% accuracy for $\lambda = 0.5$. Additionally, this parameter seems to help the model generalize more effectively, since the accuracy of all three of the training, validation, and test data seems to be closer throughout every single epoch, especially for larger values of λ.

The final classification accuracy for both $\lambda = 0.1$ and $\lambda = 0.5$, for all three data sets, lies between 96.5% and 97.25%, clearly indicating a better generalization. For the case when $\lambda = 0.001$, the performance of the model on the training set is 78.08%, and decreases to approximately 70% for the validation and test data sets.

A possible use for this parameter is to begin with it set close to zero to begin. If the accuracy on the training data is exceeding the accuracy on the validation set by some threshold, then the regularization parameter can be increased, which would likely increase the generalization of the model. Finally, the regularization parameter appears to have little impact on the time required to train the model; the required time was between 16 and 18 seconds when training for all three different values of λ.

1.5 Comparing Batch Gradient Descent with Normal Equation

Analytically, the weight vector which will result in the smallest squared error is determined by taking the gradient of the cost function. Defining our cost function as $\mathcal{F} = ||\underline{y} - \mathcal{X}\underline{\mathbf{w}}||^{\mathbf{2}}$, the gradient is calculated below.

$$\begin{aligned}
\mathcal{F}_{MSE}(\underline{w}) &= ||\underline{y} - \boldsymbol{\mathcal{X}}\underline{w}||^2 = (\underline{y} - \boldsymbol{\mathcal{X}}\underline{w})^T(\underline{y} - \boldsymbol{\mathcal{X}}\underline{w}) \\
&= \underline{y}^T\underline{y} - 2\underline{w}^T\boldsymbol{\mathcal{X}}^T\underline{y} + \underline{w}^T\boldsymbol{\mathcal{X}}^T\boldsymbol{\mathcal{X}}\underline{w} \\
\nabla_{\underline{w}^T}\mathcal{F}_{MSE}(\underline{w}) &= -2\boldsymbol{\mathcal{X}}^T\underline{y} + 2\boldsymbol{\mathcal{X}}^T\boldsymbol{\mathcal{X}}\underline{w} = 0 \\
\therefore \boldsymbol{\mathcal{X}}^T\underline{y} &= \boldsymbol{\mathcal{X}}^T\boldsymbol{\mathcal{X}}\underline{w} \\
\therefore \underline{w}^\star &= (\boldsymbol{\mathcal{X}}^T\boldsymbol{\mathcal{X}})^{-1}\boldsymbol{\mathcal{X}}^T\underline{y}
\end{aligned}$$

Note that when a regularization term is included in the loss function, the analytical expression is slightly modified is given by:

$$\underline{w}^\star = (\boldsymbol{\mathcal{X}}^T\boldsymbol{\mathcal{X}} + \lambda\mathbb{I})^{-1}\boldsymbol{\mathcal{X}}^T\underline{y}$$

The final accuracies and loss values, as well as computation time, are summarized in table 1 below. The normal equation takes two orders of magnitude less time to calculate the optimal weights when compared to gradient descent, and achieves a higher accuracy and lower loss with these weights when classifying the images in the training set.

With that being said it seems as though the normal equation is the best way to go. However, this is only the case with a relatively small dataset, such as the notMNIST dataset used, since taking the inverse of matrices that have millions of entries becomes difficult and time consuming, and the matrix may not be invertible at all.

	Computation Time	**Final Accuracy**	**Final Loss**
Batch Gradient Descent	19.99 seconds	53.77%	0.31
Normal Equation	0.19 seconds	99.31%	0.01

Table 1: Final training accuracy, loss and computation time for batch gradient descent using MSE cost function and an analytical expression for optimized weights (normal function) when the learning rate with no weight decay.

2 Logistic Regression

2.1 Loss Function and Gradient

The Cross-Entropy error function is less sensitive to mislabelled training examples and outliers, and thus, is a more suitable loss function for the classification task. We only care about the probability of a data point belonging to one class, so the real valued linear prediction is fed through a sigmoid function and squashed so as to lie between 0 and 1. The sigmoid function is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{5}$$

The Cross-Entropy Loss function, with an additional regularization term, is given by:

$$\mathcal{F}_{CE} = \sum_{n=1}^{N} \frac{1}{N}\Big[- y^{(n)} \log \hat{y}(\mathbf{x}^{(n)}) - (1 - y^{(n)}) \log (1 - \hat{y}(\mathbf{x}^{(n)}))\Big] + \frac{\lambda}{2}||\underline{w}||_2^2 \tag{6}$$

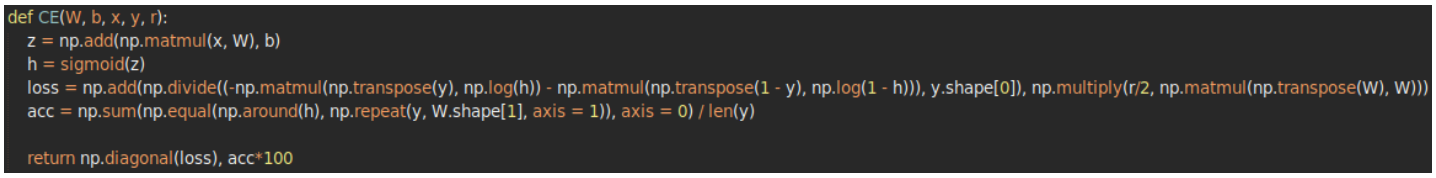
The vectorized version of this loss function is given by:

$$\mathcal{F}_{CE} = \frac{(-\underline{y}^T \log \underline{h} - (1-\underline{y})^T \log(1-\underline{h}))}{N} + \frac{\lambda \underline{w}^T \underline{w}}{2} \tag{7}$$

The gradient of this function is nearly identical to the MSE cost function, with just the sigmoid function used to predict the labels:

$$\frac{\partial \mathcal{F}_{CE}}{\partial b} = \frac{\{\sigma(\boldsymbol{x}^T \underline{w} + \underline{b}) - \underline{y}\}}{N} \quad , \quad \frac{\partial \mathcal{F}_{CE}}{\partial \underline{w}} = \frac{\boldsymbol{x}^T \{\sigma(\boldsymbol{x}\underline{w} + \underline{b}) - \underline{y}\}}{N} + \lambda \underline{w} \tag{8}$$

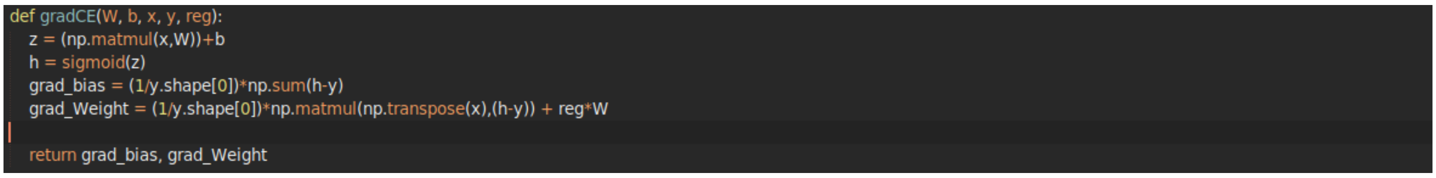
The code snippets for both the Cross-Entropy cost function and its gradients are provided below:



```
def CE(W, b, x, y, r):
    z = np.add(np.matmul(x, W), b)
    h = sigmoid(z)
    loss = np.add(np.divide((-np.matmul(np.transpose(y), np.log(h)) - np.matmul(np.transpose(1 - y), np.log(1 - h))), y.shape[0]), np.multiply(r/2, np.matmul(np.transpose(W), W)))
    acc = np.sum(np.equal(np.around(h), np.repeat(y, W.shape[1], axis = 1)), axis = 0) / len(y)

    return np.diagonal(loss), acc*100
```

Figure 7: Python code snippet of the implemented vectorized Cross-Entropy cost function.



```
def gradCE(W, b, x, y, reg):
    z = (np.matmul(x,W))+b
    h = sigmoid(z)
    grad_bias = (1/y.shape[0])*np.sum(h-y)
    grad_Weight = (1/y.shape[0])*np.matmul(np.transpose(x),(h-y)) + reg*W

    return grad_bias, grad_Weight
```

Figure 8: Python code snippet of the implemented vectorized Cross-Entropy cost function gradients.

2.2 Learning

The gradient descent function was modified to include a flag which indicates whether Cross-Entropy or MSE should be used as the loss function. The regularization term was set to $\lambda = 0.1$, the learning rate was set to $\alpha = 0.005$ and the optimization was run for 5000 epochs, generating the following two plots:

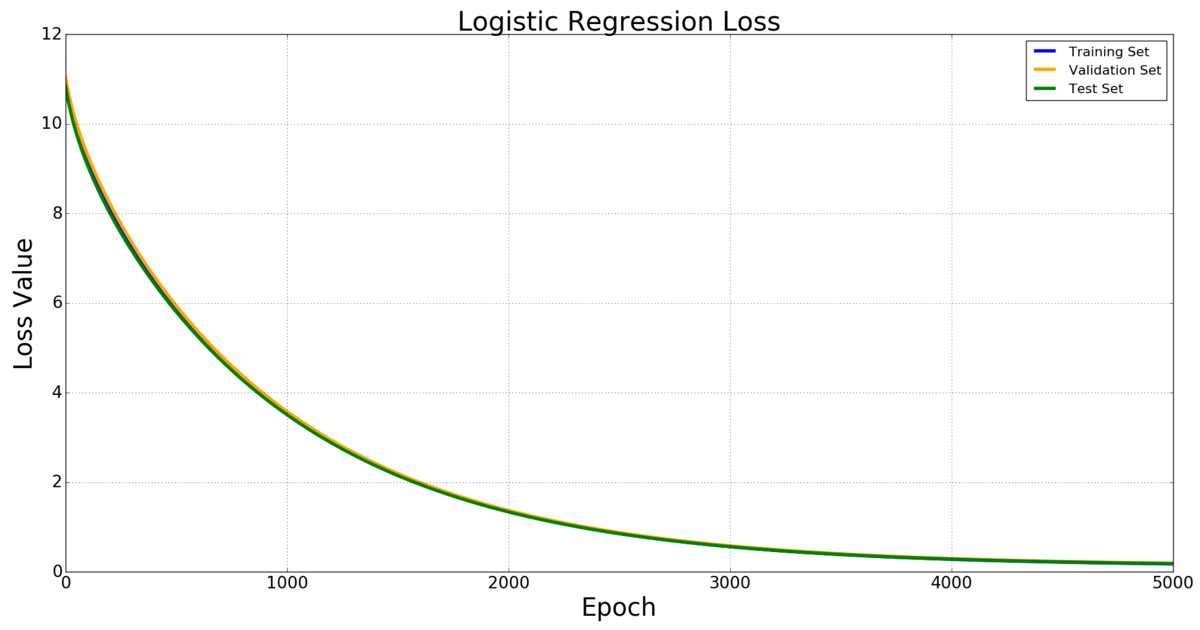


Figure 9: Calculated Cross-Entropy loss at each of 5000 training epochs, with learning rate $\alpha = 0.005$ and regularization $\lambda = 0.1$.

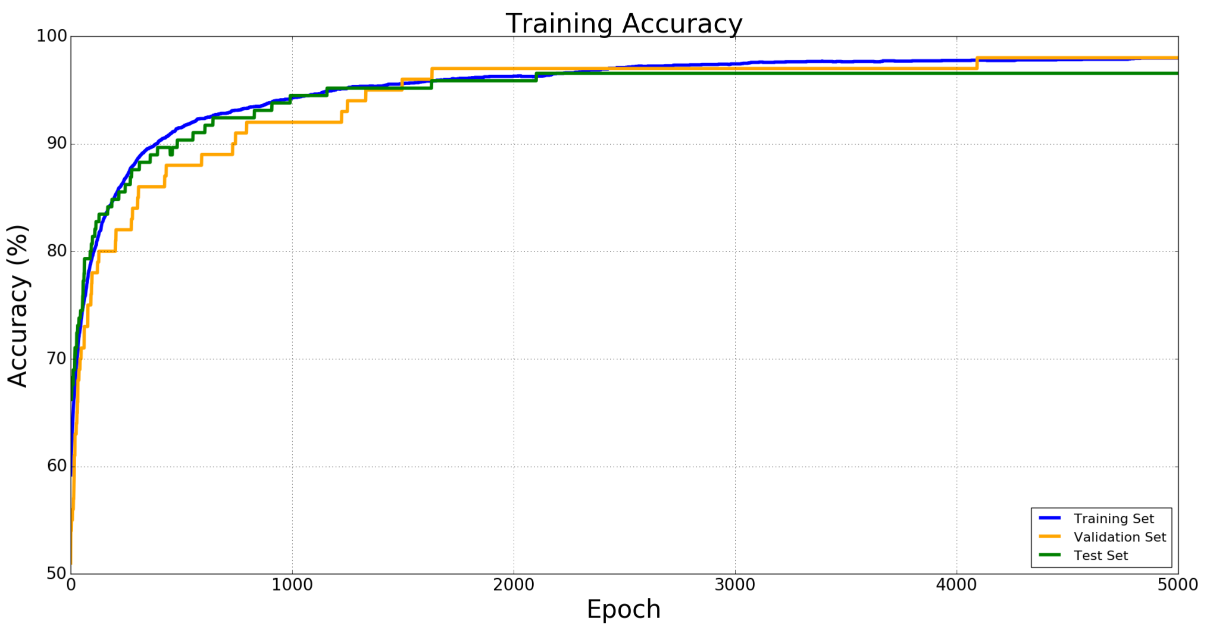


Figure 10: Calculated Cross-Entropy accuracy at each of 5000 training epochs, with learning rate $\alpha = 0.005$ and regularization $\lambda = 0.1$.

Clearly, The Cross-Entropy loss function does a better job at the bi-linear classification task. For the same parameters α and λ, the CE Loss function achieves 96-98% accuracy compared to the 96% achieved by the MSE loss function. This isn't a major improvement, but any improvement is important when building machine learning models. The time required to train the model was roughly the same, on the order of 15 seconds, so there is no improvement in training time.

2.3 Comparison to Linear Regression

The optimization of the notMNIST dataset was performed with both loss functions and plotted below as per the parameter specifications required by the question.

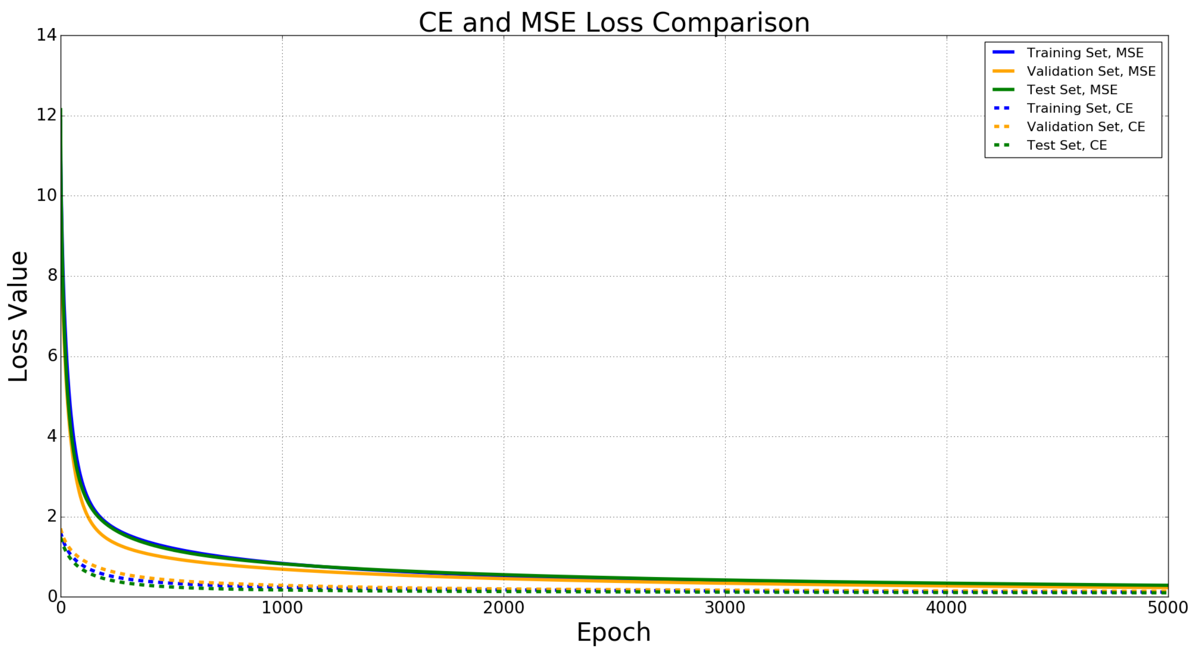


Figure 11: Calculated Cross-Entropy and MSE loss at each of 5000 training epochs, with learning rate $\alpha = 0.005$ and regularization $\lambda = 0.0$.

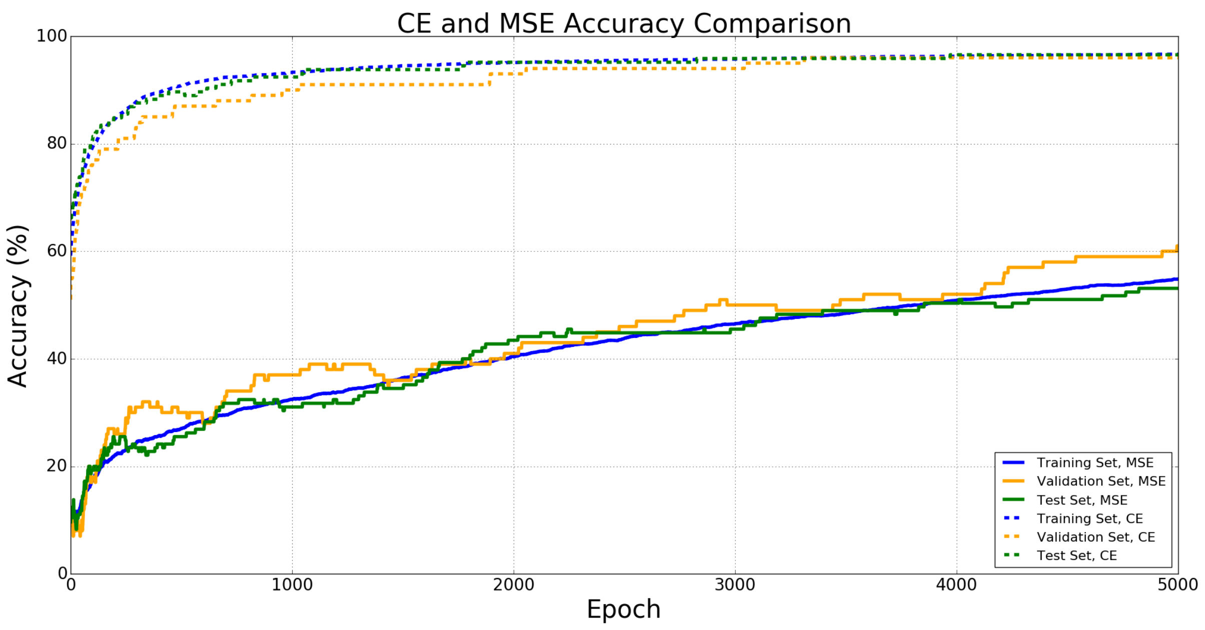


Figure 12: Calculated Cross-Entropy and MSE accuracy at each of 5000 training epochs, with learning rate $\alpha = 0.005$ and regularization $\lambda = 0.0$.

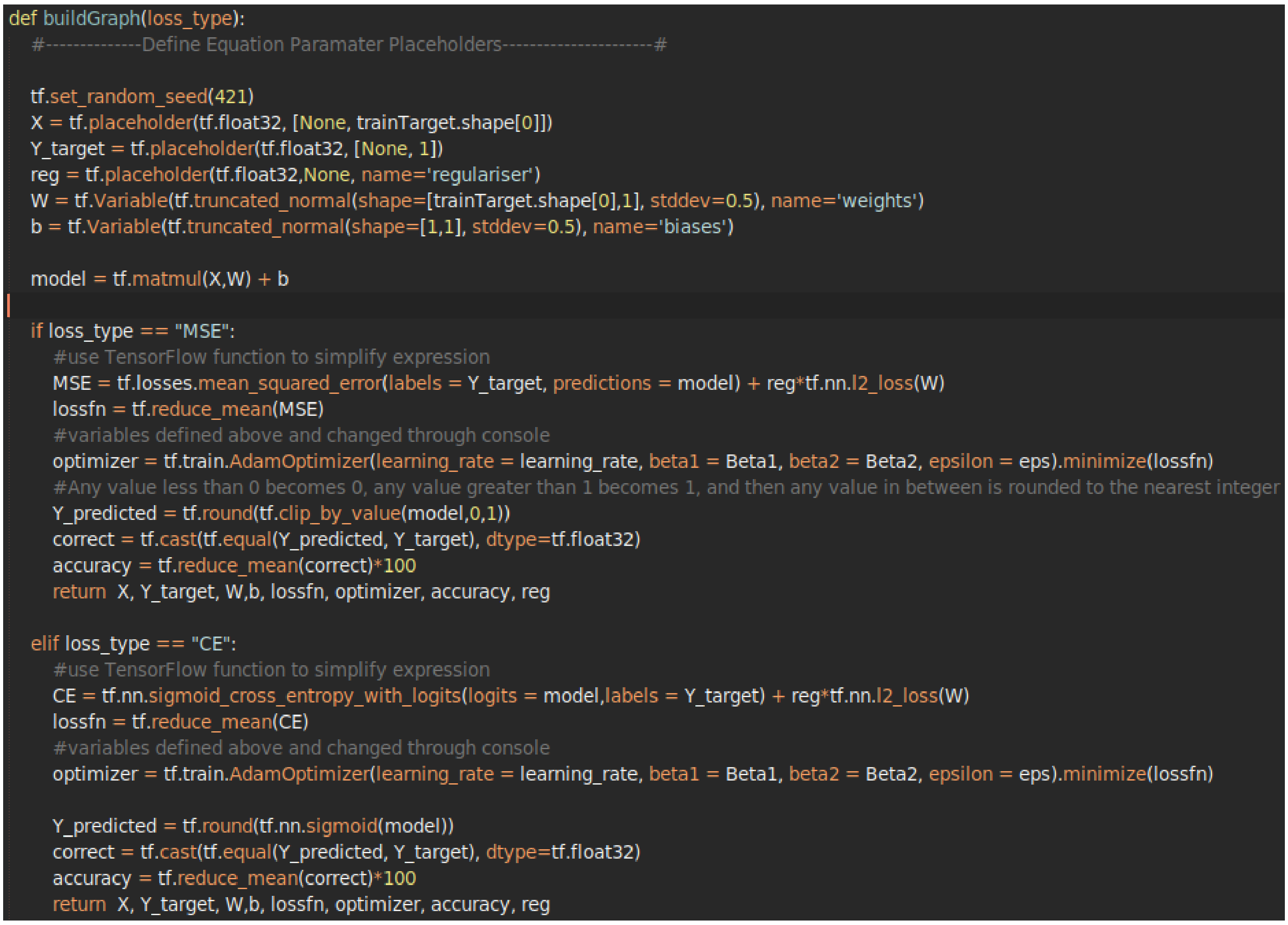
These figures prove definitively that the Cross-Entropy loss function is the correct choice for this type of problem. Not only does it achieve an accuracy in excess of 95% compared to the approximately 55-60% achieved with MSE for all 3 of the training, validation and test datasets, it also reaches convergence at this value within half the number of epochs used. While MSE might get us to a similar accuracy eventually, Cross-Entropy does it much more efficiently.

3 Batch Gradient Descent vs. SGD and Adam

In the exercises above, a batch gradient descent algorithm was implemented. For large datasets however, obtaining the gradient for all of the training data is typically unfeasible. Stochastic Gradient Descent will be used in solving this problem. Let's implement this process using the Adaptive Moment Estimation Technique (ADAM), using TensorFlow.

3.1 Building the Computational Graph

The code snippet below shows the implementation of the of the *buildGraph()* function. The code initializes the parameters, and defines the model variables which will be used when we run a session.



```python
def buildGraph(loss_type):
    #-------------Define Equation Paramater Placeholders---------------------#

    tf.set_random_seed(421)
    X = tf.placeholder(tf.float32, [None, trainTarget.shape[0]])
    Y_target = tf.placeholder(tf.float32, [None, 1])
    reg = tf.placeholder(tf.float32,None, name='regulariser')
    W = tf.Variable(tf.truncated_normal(shape=[trainTarget.shape[0],1], stddev=0.5), name='weights')
    b = tf.Variable(tf.truncated_normal(shape=[1,1], stddev=0.5), name='biases')

    model = tf.matmul(X,W) + b

    if loss_type == "MSE":
        #use TensorFlow function to simplify expression
        MSE = tf.losses.mean_squared_error(labels = Y_target, predictions = model) + reg*tf.nn.l2_loss(W)
        lossfn = tf.reduce_mean(MSE)
        #variables defined above and changed through console
        optimizer = tf.train.AdamOptimizer(learning_rate = learning_rate, beta1 = Beta1, beta2 = Beta2, epsilon = eps).minimize(lossfn)
        #Any value less than 0 becomes 0, any value greater than 1 becomes 1, and then any value in between is rounded to the nearest integer
        Y_predicted = tf.round(tf.clip_by_value(model,0,1))
        correct = tf.cast(tf.equal(Y_predicted, Y_target), dtype=tf.float32)
        accuracy = tf.reduce_mean(correct)*100
        return  X, Y_target, W,b, lossfn, optimizer, accuracy, reg

    elif loss_type == "CE":
        #use TensorFlow function to simplify expression
        CE = tf.nn.sigmoid_cross_entropy_with_logits(logits = model,labels = Y_target) + reg*tf.nn.l2_loss(W)
        lossfn = tf.reduce_mean(CE)
        #variables defined above and changed through console
        optimizer = tf.train.AdamOptimizer(learning_rate = learning_rate, beta1 = Beta1, beta2 = Beta2, epsilon = eps).minimize(lossfn)

        Y_predicted = tf.round(tf.nn.sigmoid(model))
        correct = tf.cast(tf.equal(Y_predicted, Y_target), dtype=tf.float32)
        accuracy = tf.reduce_mean(correct)*100
        return  X, Y_target, W,b, lossfn, optimizer, accuracy, reg
```

Figure 13: Python code snippet of the *buildGraph()* function

3.2 Implementing SGD

In this section, we minimize the MSE and compare different values of $\lambda = \{0, 0.1, 0.5\}$ over 700 epochs, with a batch size of 500. The learning rate is set to $\alpha = 0.001$. Below, the graphs show the training, validation, and test sets' loss and accuracy plots for the different values of λ.

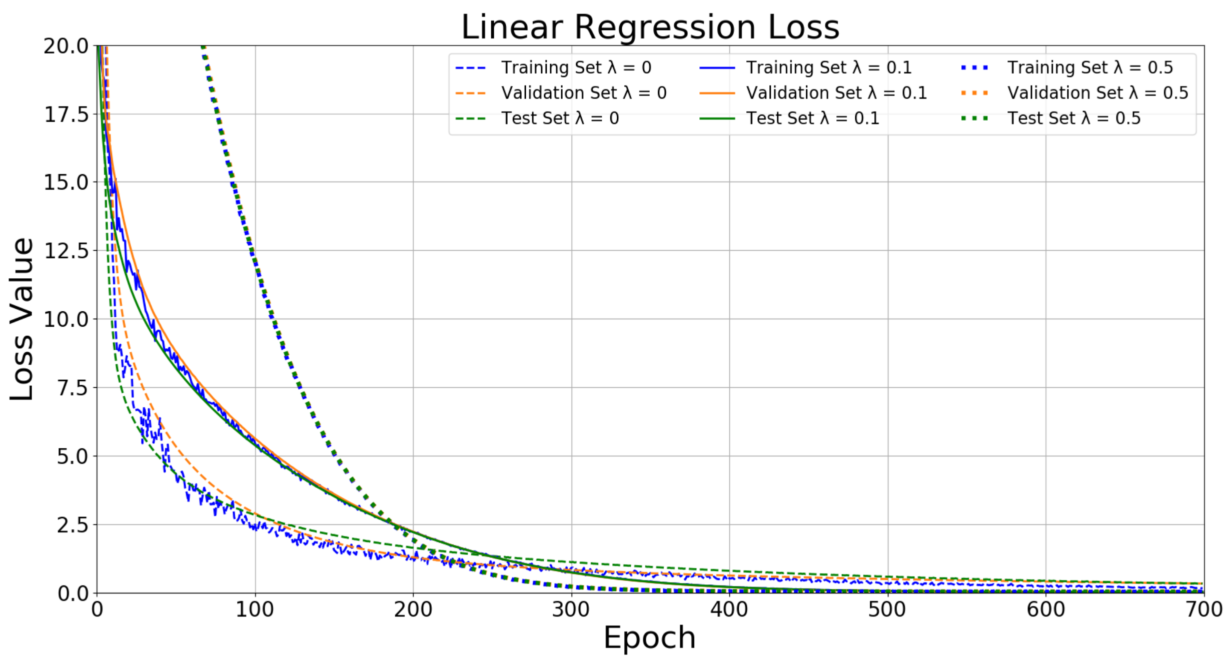


Figure 14: Loss Plots for λ

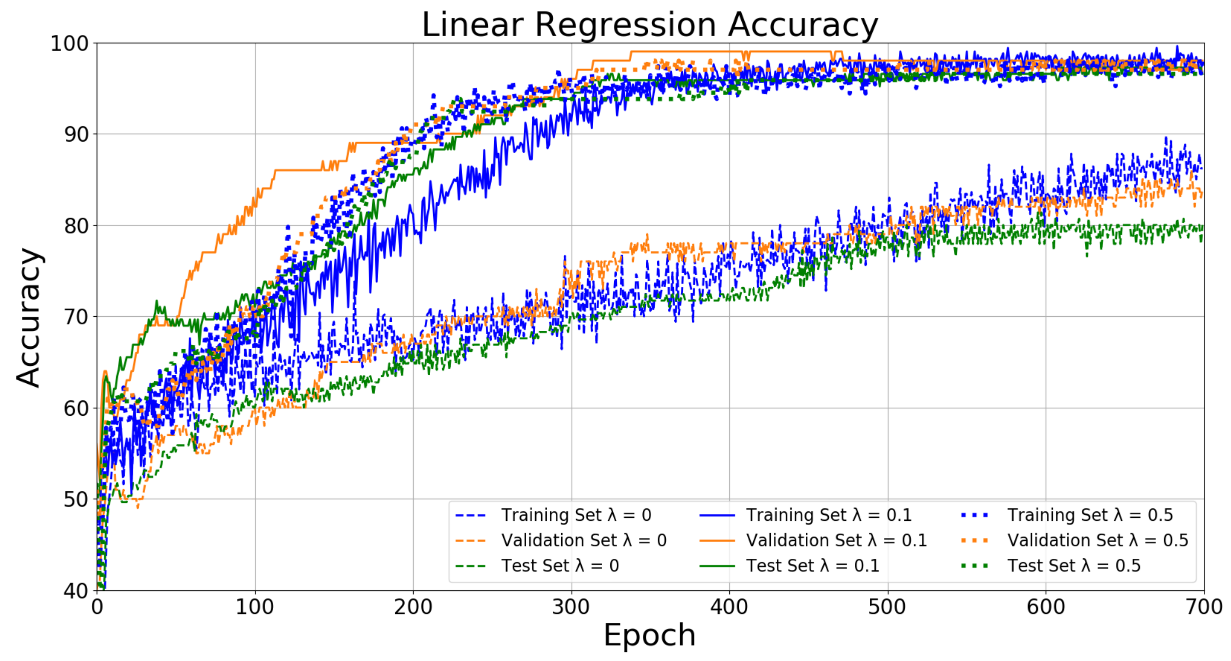


Figure 15: Accuracy Plots for λ

It can be seen that increasing λ decreases the difference between the training, validation, and test set errors, and increases the accuracy. In addition, increasing the regularization parameter decreases the time needed for convergence. In Figure 14, the higher the value of λ the steeper the curve.

When λ is set to zero, the accuracy of the final test results is around 80%. Whenλ's value increases, the accuracy jumps to 96%. This big jump shows the weakness of using linear regression as a classifier. In the binary classification scenario, it can probably classify well only if λ is used, meaning the model usually under-fits the data, and only a boost in regularization helps it classify letters better.

3.3 Batch Size Investigation

In this section, we study the effects of changing the batch size on the behaviour of the SGD algorithm. We tested on batch sizes equal to $B = \{100, 700, 1750\}$. The graphs below the loss and accuracy curves how for each B. In this section and onward, we will keep α and λ set at 0.001 and 0.1 respectively.

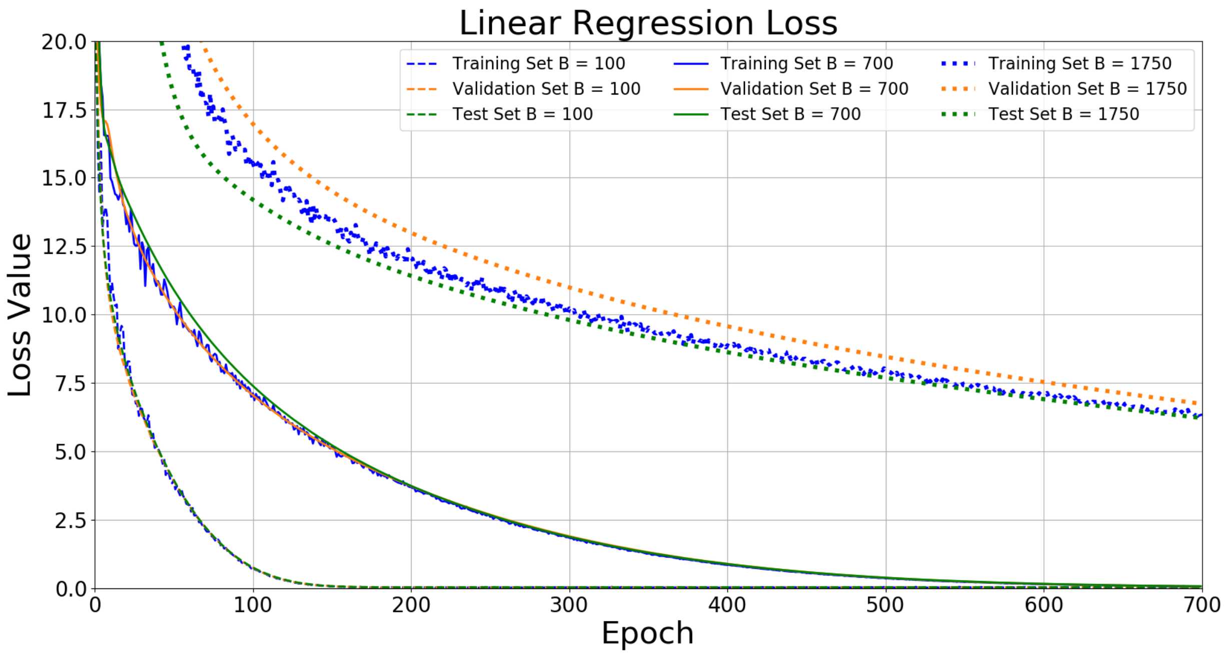


Figure 16: Loss Plots for B

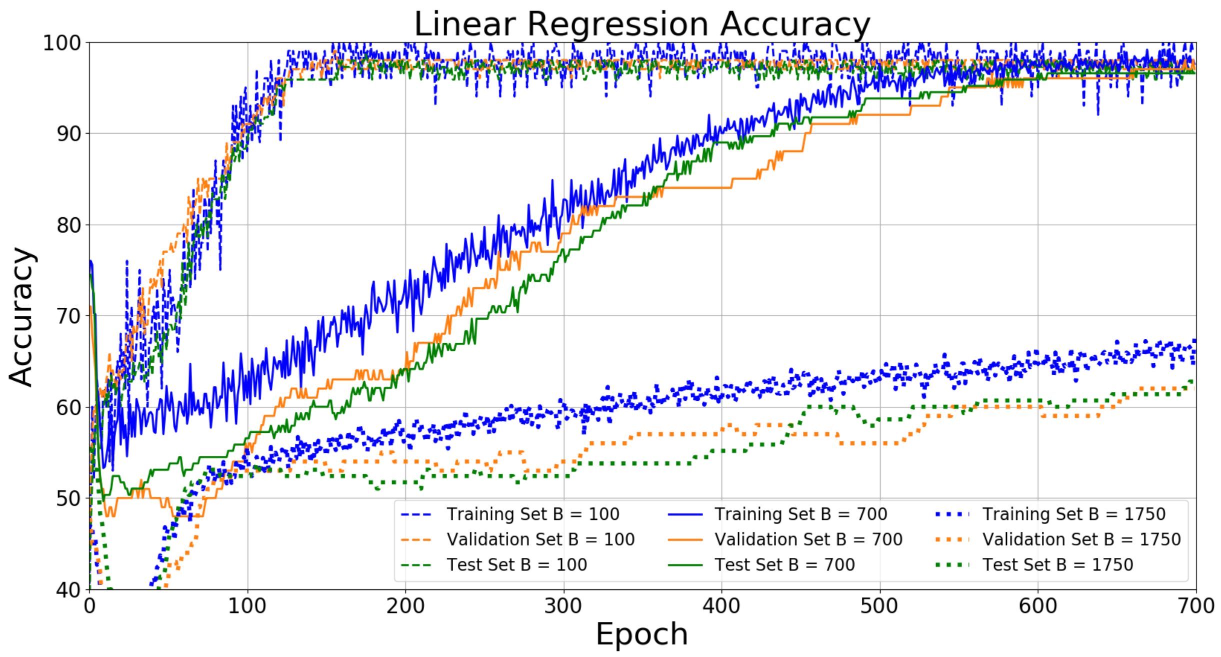


Figure 17: Accuracy Plots for B

A difference between the 3 graphs is the noisiness of each step. This decreases as B increases. The noisiness results from using less data to find the direction of the maximum gradient at each step. So we trade taking the most optimal step for quicker convergence. This is seen in both pits as the larger the value of B, the loss is higher and the accuracy is lower compared to smaller values of B. An interesting note is if the batch size was equal to the training set, then we'd have gradient descent like in Part 1, where all the set is used to calculate the gradient at each epoch.

3.4 Hyper-parameter Investigation

In this section, we investigate the effect of changing the ADAM optimizer hyper-parameters, $\beta1$, $\beta2$, and ϵ, on the loss and accuracy of the linear regression model.

Changing $\beta1$ values The $\beta1$ parameter is defined as the rate of exponential decay of the first momentum term in the ADAM Optimizer [1]. This means the larger the value of $\beta1$'s, the less it will use previous estimates to calculate the gradient. This likely means a slower rate of convergence. The final test accuracy when $\beta1$ was set to 0.95 and 0.99 was 97.2% and 91.72% respectively.

Changing $\beta2$ values

The $\beta2$ parameter is defined as the rate of exponential decay of the second momentum term in the ADAM Optimizer [1]. It's relationship to the learning rate is as follows:

$$\text{Learning Rate} \propto \sqrt{1-\beta2}$$

This means as $\beta2$ increases, the learning rate decreases as the square root term will approach zero. A slower learning rate means the rate of convergence will be slower. This should be seen as a smaller decrements in loss from epoch to epoch. On the final test result, for the case of linear regression, setting $\beta2$ to 0.99 or 0.9999 had no affect on the final test accuracy; it remained at 97.24%.

Changing ϵ values

The ϵ variable is used to stabilize the numerical calculation [1]. A higher value of ϵ can be seen as a reduction in noisiness when error and accuracy are plotted vs epochs.Varying this had no affect on the final test accuracy, it remained at 97.24%.

3.5 Cross Entropy Loss Investigation

In this section we compare Logistic Regression predictions using TensorFlow to the Linear Regression case. The same exercises are repeated, where we will show the variance of loss and accuracy vs epochs as we vary the regularization term, the batch size, and the hyper-parameters of the ADAM optimizer.

As opposed to Linear Regression where we are trying to fit a line through all the data points, the calculated number from the Logistic model has a meaning: it is the probability that prediction is the letter C or J. We select 0.5 as our confidence boundary. If the probability is greater than 0.5, then the model will predict the positive classifier (letter 'C') and if it's less than 0.5, then it will predict 0 (the letter 'J'). Using probability is more suited for the case of classification, as we shall see in all the accuracy curves below; the Logistic model is a better predictor than the Linear Regression one.

3.5.1 Implementing SGD

In this section, we minimize the Cross Entropy Loss and compare different values of $\lambda = 0, 0.1, 0.5$ over 700 epochs, with a batch size of 500. The learning rate is set to $\alpha = 0.001$. Below, the graphs show the training, validation, and test sets loss and accuracy plots for the different values of λ.

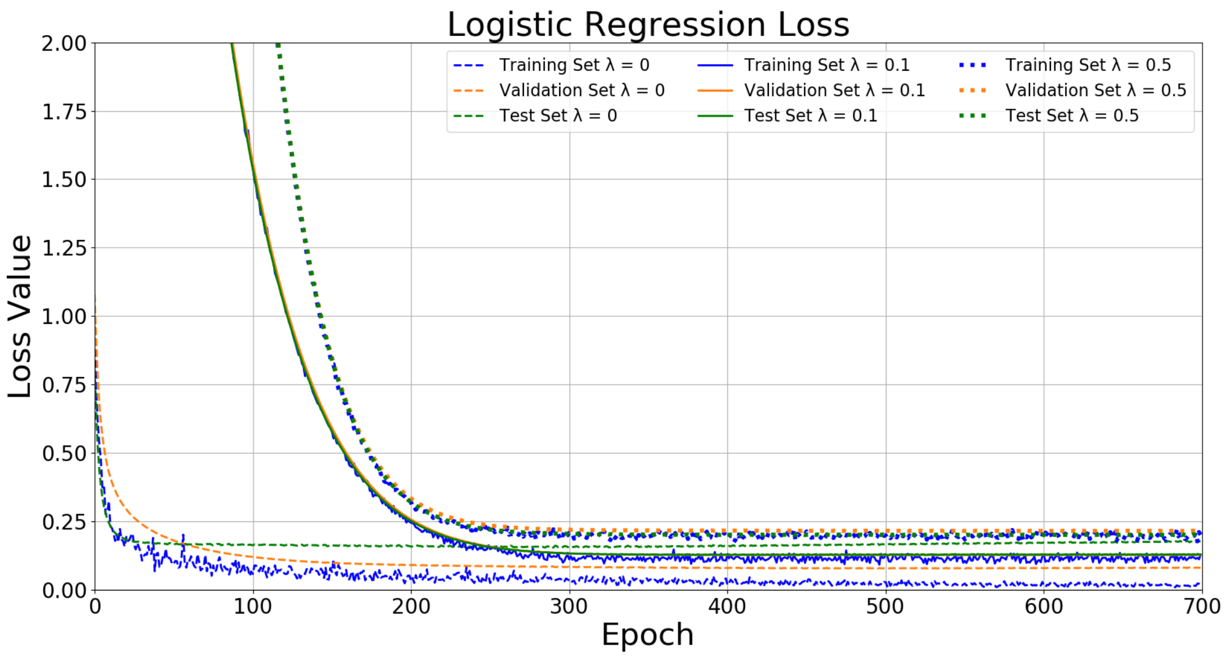


Figure 18: Loss Plots for λ

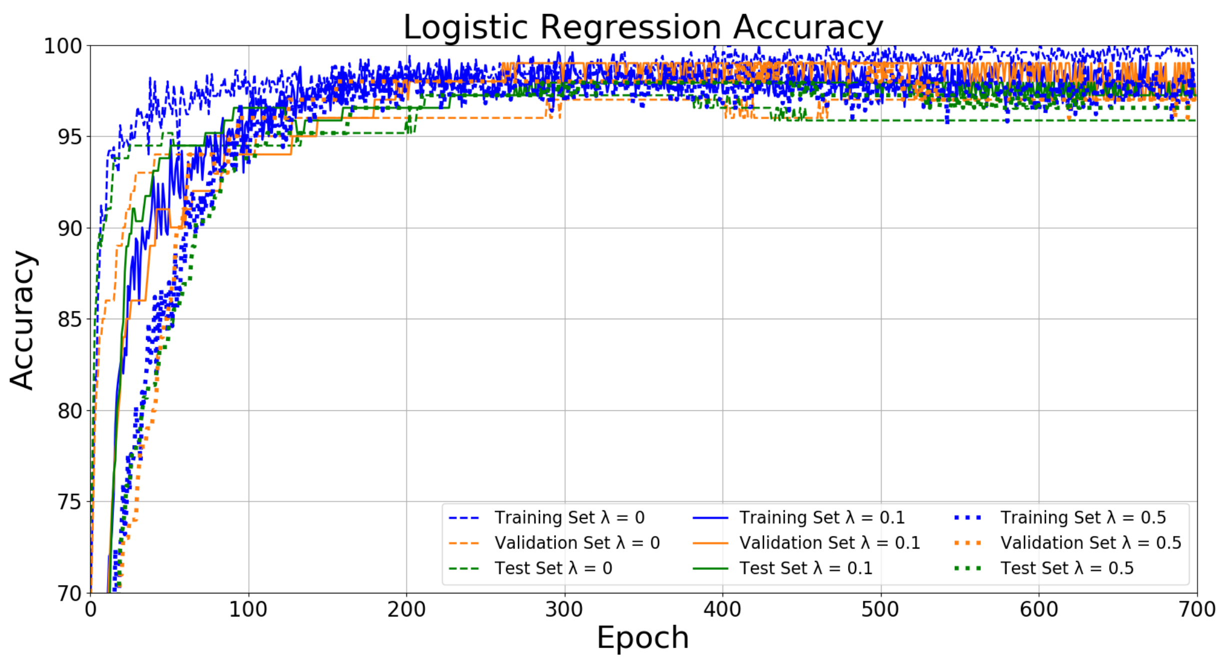


Figure 19: Accuracy Plots for λ

The importance of having a regularization term λ in logistic regression is shown in the figures. When λ is zero, it seems like the small loss on the training and validation set would mean the test set would be the same; however the Loss Plot shows that is not the case. The test set error does not reduce after around 100 epochs. When λ is included, the test set error is nearly identical to the training and validation set, as the model no longer over-fits the predicted values.

A stark difference between logistic and linear regression can be seen from the accuracy plots: logistic regression's accuracy predictions are in the region of 95%+ for all the values of λ , whereas linear regression's accuracy was near 80% when λ was 0.

3.5.2 Batch Size Investigation

In this section, we study the effects of changing the batch size on the behaviour of the SGD algorithm. We tested on batch sizes equal to $B = \{100, 700, 1750\}$. The graphs below show for each B, the loss and accuracy curves. In this section and onward, we will keep α and λ set at 0.001 and 0.1 respectively.

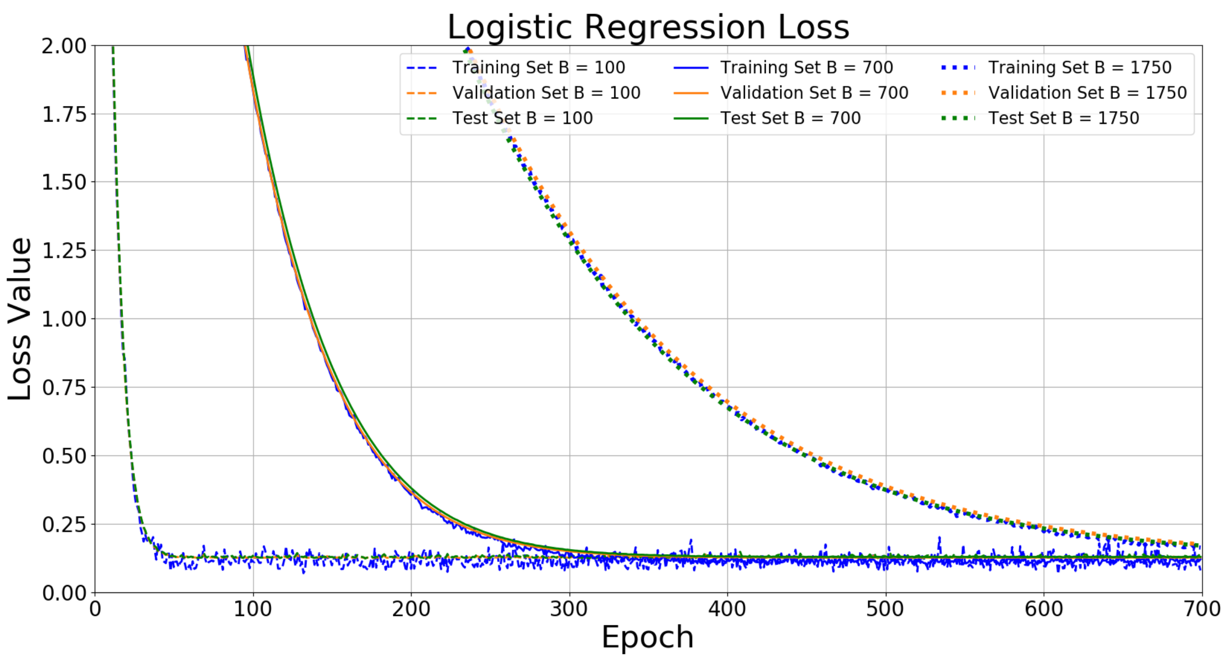


Figure 20: Loss Plots for B

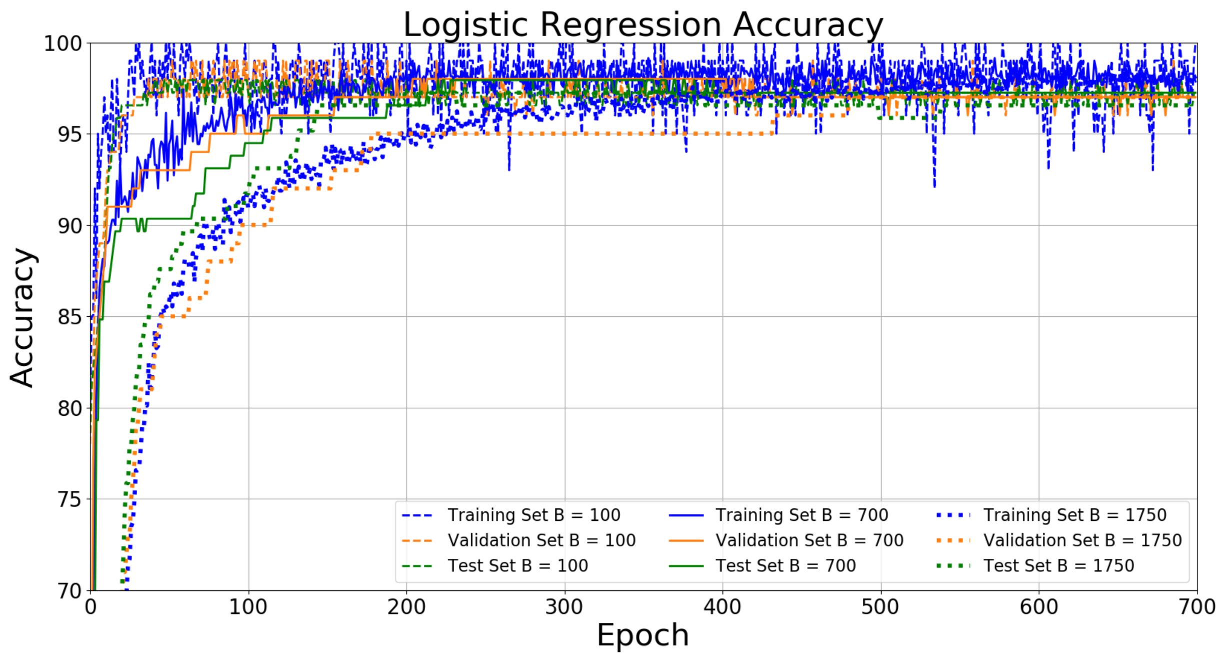


Figure 21: Accuracy Plots for B

As discussed in the Linear Regression case, decreasing the batch size trades smoothness of descent with quicker convergence, and vice-versa when the batch size increases. Decreasing the batch size influences the variance of accuracy predictions from epoch to epoch.

3.5.3 Hyper-parameter Investigation

In this section, we investigate the effect of changing the ADAM optimizer hyper-parameters, $\beta1$, $\beta2$, and ϵ, on the loss and accuracy of the linear regression model.

Changing $\beta1$ values

The $\beta1$ parameter is defined as the rate of exponential decay of the first momentum term in the ADAM Optimizer [1]. This means the larger the value of $\beta 1$'s, the less it will use previous estimates to calculate the gradient. This likely means a slower rate of convergence. In the case of Logistic Regression, the value of $\beta 1$ did not change the final test accuracy value of 97.24%. This means that the model has likely converged and stabilized within the first 700 epochs.

Changing $\beta 2$ values On the final test result, for the case of logistic regression, setting $\beta 2$ to 0.99 or 0.9999 had no affect on the final test accuracy; it remained at 97.24%. This means the impacts of this second momentum term where reached prior to the 700th epoch.

Changing ϵ values The ϵ variable is used to stabilize the numerical calculation [1]. A higher value of ϵ can be seen as a reduction in noisiness when error and accuracy are plotted vs epochs. Unlike in Linear Regression, varying ϵ between 10^{-4} and 10^{-9} changed the final test set accuracy from 97.24% to 97.93%. This may seem like a small difference, but in classification, every percent point matters!

3.6 Comparison Against Batch Gradient Descent

The biggest advantage of using ADAM over batch gradient descent is the speed at each time step, especially when a small batch-size is used. The losses and accuracy of the final model are nearly identical to each other. What is sacrificed when SGD is implemented is that most optimal path of descent at each time step is not chosen. In batch descent, the algorithm uses all the training set points at every time step to calculate the path to descend, whereas in SGD, it's only based on the number of points in the batch. As SGD takes a less direct approach to the minimum, it results in the spikes in the loss curves plotted.

Even though SGD takes a longer route to reach the minimum, it does so faster than using the more direct method of batch descent. This is analogous to taking the 407 on a Friday afternoon in the summer to go around the city during rush hour as opposed to using the more direct, but much, much slower 401.

In general, it seems like the dataset of each C and J both had very low variance, and are easily separable from each other. We were expecting the linear regression model to perform much worse than it did as it tries to fit a straight line through all the points. If the dataset were to compromise of more than 2 letters, it seems highly unlikely that linear regression will perform as well as logistic regression.

References

[1] TensorFlow, *tf.train.AdamOptimizer — TensorFlow*, 2019 [Accessed: 04- Feb- 2019].