ECE421
Assignment 1
Linear and Logistic Regression

David Larsson, Robert Agnoletto

January 2019

note: 50/50 contribution between partners.

1 Linear Regression

1.1 Loss function and the gradient

1.1.1 Analytic Expression

The loss function in this assignment is given by adding the Mean Squared Error and the weight decay loss, this is shown in equation 1 shown below.

$$\begin{aligned}\mathcal{L} &= \mathcal{L}_D + \mathcal{L}_W \\ &= \sum_{n=1}^{N} \frac{1}{2N} ||\boldsymbol{W}^T \boldsymbol{x}^{(n)} + b - y^{(n)}||_2^2 + \frac{\lambda}{2} ||\boldsymbol{W}||_2^2 \end{aligned} \tag{1}$$

Using this expression the whole gradient for the loss function can be derived by taking gradient with regards to the weight, $\boldsymbol{W}$ and then the gradient with regards to the bias, b. This is given in the equations below

$$\boldsymbol{\nabla_w(\mathcal{L})} = \sum_{n=1}^{N} \frac{1}{N} ||\boldsymbol{W}^T \boldsymbol{x}^{(n)} + b - y^{(n)}||_2 \boldsymbol{x}^{(n)} + \lambda \boldsymbol{W} \tag{2}$$

$$\nabla_b(\mathcal{L}) = \sum_{n=1}^{N} \frac{1}{N} ||\boldsymbol{W}^T \boldsymbol{x}^{(n)} + b - y^{(n)}||_2 \tag{3}$$

1.1.2 Python code

```
def MSE(W,b,x,y,reg):
    N = np.size(y,0)
    mse = 1/(2*N)*npl.norm(np.matmul(x,W) + b - y)**2+reg/2*npl.norm(W)**2
    return mse

def grad_MSE(W,b,x,y,reg):
    N = np.size(y,0)
    nablaw = np.matmul(np.transpose(x),(np.matmul(x,W) + b - y))/N + W*reg
    nablab = np.mean(np.matmul(x,W) + b - y)
    return nablaw, nablab
```

1.2 Tuning the Learning Rate

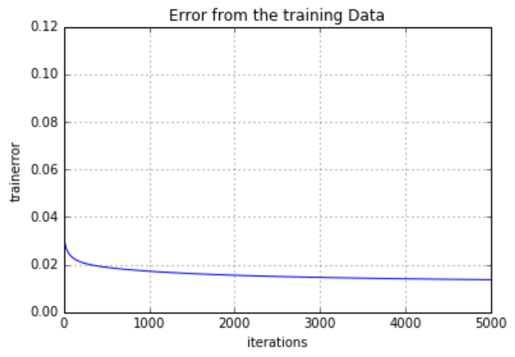

Figure 1: The MSE from the training data generated using linear regression $\alpha = 0.005$.

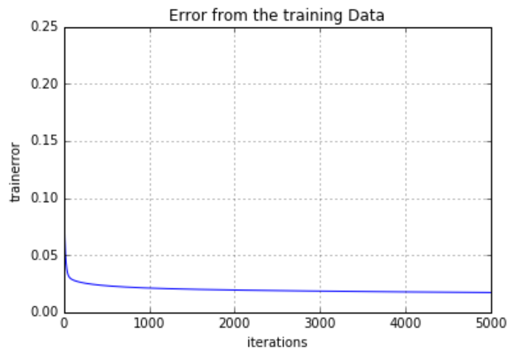

Figure 2: The MSE from the training data generated using linear regression $\alpha = 0.001$.

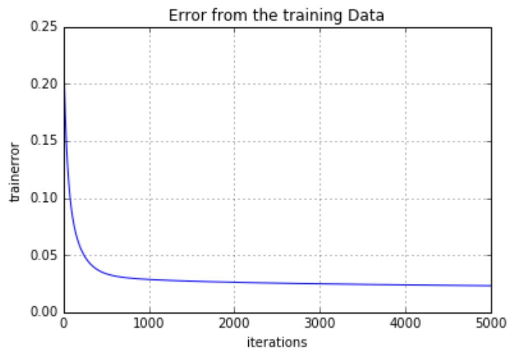

Figure 3: The MSE from the training data generated using linear regression $\alpha = 0.0001$.

α	MSE of training data	MSE of validation data	MSE of test data
0.005	0.01364	0.01458	0.01776
0.001	0.01722	0.01715	0.01995
0.0001	0.02316	0.02094	0.02381

Table 1: The MSE from using Linear Regression changing the learning rate, α.

In figures 1, 2 and 3 one can observe the training error using the weights calculated through the iterations of the gradient descent algorithm. From looking at theses plots and noticing the scale at the y-axis one can observe that the lower the learning rate the slower the convergence seems to become. This is expected but in the case where the function could overshoot the minimum a higher learning rate might not be beneficial however this does not seem to be the case here. Also looking at table 1 the final Mean Squared error of the training, validation and test data is shown. From these one can conclude that the higher learning rates did also converge to a lower mean squared error for all of the different data sets.

1.3 Generalization

λ	MSE of training data	MSE of validation data	MSE of test data
0.001	0.01371	0.01464	0.01779
0.1	0.01673	0.01723	0.01889
0.5	0.02047	0.02038	0.02067

Table 2: The Mean Squared error of the training, validation and test data using the final optimized weight with different regularization parameter, λ.

From table 2 one can conclude increasing the regularization parameter, λ, the mean squared error of all the data sets seems to increase. However in the case of $\lambda = 0.5$ the mean squared error actually is lower for validation data then for train data and also the difference between the different sets in general seems to be lower for larger regularization parameter. This shows the effect of over-modeling since the fit is better for the training data for lower λ but it does not actually describe the ”trend” better.

1.4 Comparing Batch GD with normal equation

MSE of training data	MSE of validation data	MSE of test data
0.009385	0.020427	0.025948

Table 3: The Mean Squared Error for the different data sets using normal equations to compute the optimized weights.

Comparing the results for normal equations in 3 with our previous results the MSE of the training data is actually the lowest we have observed which also is expected since this method should produce the actual minimum of the function while the gradient descent only converge towards it. Although we again have a case of over-modeling since the MSE of the validation and test data is actually the worst we have observed.

2 Logistic Regression

2.1 Binary cross-entropy loss

For this classification task we instead of using the MSE as previously we instead use the cross-entropy loss and still keeping the weight decay loss. The whole cross entropy-loss is given by equation 5 where $\hat{y}(x^{(n)}) = \sigma(\boldsymbol{W}^T\boldsymbol{x} + b)$ where $\sigma(x) = \frac{1}{1+e^{-x}}$.

2.1.1 Analytical Expressions

$$\begin{aligned}
\mathcal{L} &= \mathcal{L}_D + \mathcal{L}_W \\
&= \sum_{n=1}^{N} \frac{1}{N}\Big(- y^{(n)} \log \hat{y}(\boldsymbol{x}^{(n)}) - (1 - y^{(n)}) \log(1 - \hat{y}(\boldsymbol{x}^{(n)}))\Big) + \frac{\lambda}{2}||\boldsymbol{W}||_2^2 \qquad (4) \\
&= \sum_{n=1}^{N} \frac{1}{N}\Big(- y^{(n)}(\boldsymbol{W}^T\boldsymbol{x}^{(n)} + b) + \log(1 + e^{(\boldsymbol{W}^T\boldsymbol{x}^{(n)}+b)})\Big) + \frac{\lambda}{2}||\boldsymbol{W}||_2^2 \qquad (5)
\end{aligned}$$

The gradient with regards to the weights, $\boldsymbol{W}$ is then given by equation 6 and the gradient with regards to the bias, b by equation 7

$$\begin{aligned}
\boldsymbol{\nabla_w(\mathcal{L})} &= \sum_{n=1}^{N} \frac{1}{N}\Big(- y^{(n)}\boldsymbol{x}^{(n)} + \frac{\boldsymbol{x}^{(n)}e^{(\boldsymbol{W}^T\boldsymbol{x}^{(n)}+b)}}{1 + e^{(\boldsymbol{W}^T\boldsymbol{x}^{(n)}+b)}}\Big) + \lambda\boldsymbol{W} \\
&= \sum_{n=1}^{N} \frac{1}{N}\Big(- y^{(n)}\boldsymbol{x}^{(n)} + \frac{\boldsymbol{x}^{(n)}}{1 + e^{-(\boldsymbol{W}^T\boldsymbol{x}^{(n)}+b)}}\Big) + \lambda\boldsymbol{W} \qquad (6)
\end{aligned}$$

$$\nabla_b(\mathcal{L}) = \sum_{n=1}^{N} \frac{1}{N}\Big(- y^{(n)} + \frac{1}{1 + e^{-(\boldsymbol{W}^T\boldsymbol{x}^{(n)}+b)}}\Big) \qquad (7)$$

2.1.2 Python Code

```
def crossEntropyLoss(W, b, x, y, reg):
    CEloss = np.mean(-(y*(np.matmul(x,W)+b))+np.log(1+np.exp((np.matmul(x,W)+b))))
    return CEloss + reg/2*npl.norm(W)**2

def grad_CE(W, b, x, y, reg):
    gradW = np.matmul(x.T,(-(y-1/(1+np.exp(-(np.matmul(x,W)+b))))))/len(y) +reg*W
    gradB = np.mean(-(y - 1/(1+np.exp(-(np.matmul(x,W)+b)))))
    return gradW,gradB

def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol, lossType=''):
    loss, accuracy = np.zeros(epochs), np.zeros(epochs)
    if lossType == 'CE':
        for i in range(epochs):
            gt = grad_CE(W,b,x,y,reg)
            W = W - alpha * gt[0]
            b = b - alpha * gt[1]
            loss[i] = crossEntropyLoss(W,b,x,y,reg)
            accuracy[i] = calcAccuracy(W, b, x, y, 'Log')
            if npl.norm(alpha*gt[0]) < error_tol:
                return W,b,loss[:i],accuracy[:i]
    else:
        for i in range(epochs):
            gt = grad_MSE(W,b,x,y,reg)
            W = W - alpha * gt[0]
            b = b - alpha * gt[1]
            loss[i] = MSE(W,b,x,y,reg)
            accuracy[i] = calcAccuracy(W, b, x, y, 'Lin')
            if npl.norm(alpha*gt[0]) < error_tol:
                return W,b,loss[:i],accuracy[:i]
    return W,b,loss,accuracy

def calcAccuracy(W, b, x, y, regType='Log'):
    correct = 0
    if regType == 'Lin':
        y2 = (np.matmul(x,W) + b)
    else:
        y2 = 1/(1+np.exp(-(np.matmul(x,W) + b)))
    for i in range(len(y)):
        guess = 0
        if y2[i] > 0.5:
            guess = 1
        if guess == y[i]:
            correct += 1
    return correct/len(y)
```

2.2 Learning

In this section the Logistic Regression implemented by the code in 2.1.2 is tested with varying learning rates. The same rates are used as with Linear Regression; 0.005, 0.001, and 0.0001. All three trials use $\lambda = 0.1$ and 5000 epochs.

2.2.1 Loss and Accuracy Plots

The results are demonstrated here by plotting the CE loss as well as the accuracy against the current epoch to show the convergence behaviour. The accuracy is calculated by dividing the number of correct guesses by the total number of guesses. For logistic regression, a guess is considered correct when $P_w(y|x) > 0.5$, while for linear regression a threshold is used. Any guess over 0.5 is considered 1, and the rest is considered 0. The implementation of the accuracy can also be found in 2.1.2.

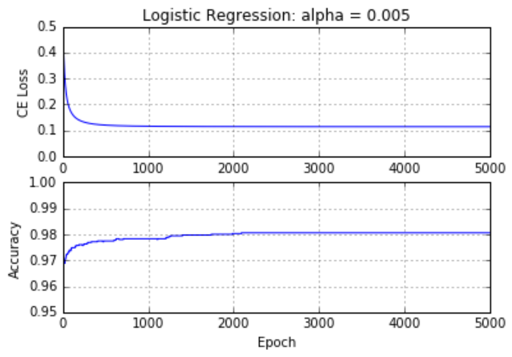

Figure 4: The CE Loss and Accuracy from the training data generated using $\alpha = 0.005$.

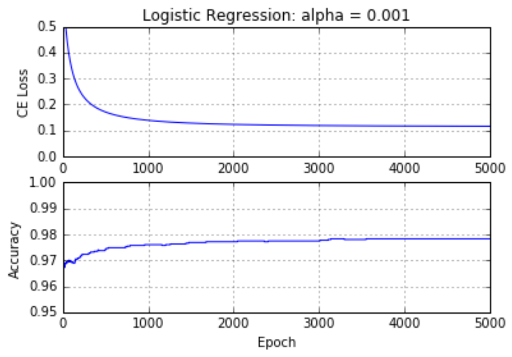

Figure 5: The CE Loss and Accuracy from the training data generated using $\alpha = 0.001$.

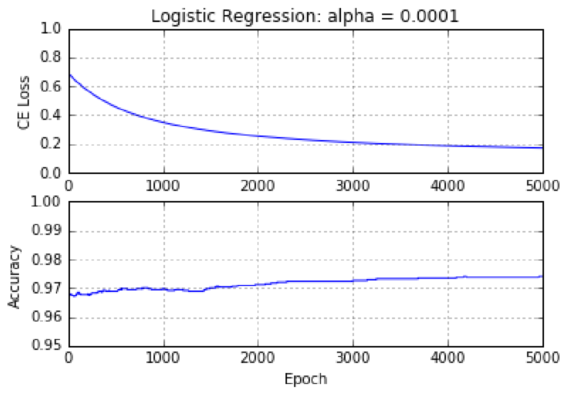

Figure 6: The CE Loss and Accuracy from the training data generated using $\alpha = 0.0001$.

As expected, the higher the learning rate, the faster the model converges. Note how in Figures 4 and 5 the CE Loss converges to around 0.1. That may seem large, but that is largely due to the weight decay loss since we set $\lambda = 0.1$. This is evident because the accuracy converges to around 0.98 for both of them which is quite high. This suggests that the loss is coming from the weights and will become more evident in Tables 4 and 5.

Notice however that in Figure 6, the learning rate of 0.0001 is too low for 5000 epochs, and the CE loss does not appear to converge fully. This is also seen in the accuracy plot which does not quite make it to 0.98 like the others did.

2.2.2 Training, Validation, and Testing Results

In addition to this, all three trained models were then used on the Training, Validation, and Testing Data. The CE losses of all three sets of data from all three models are displayed in tables. The CE loss was calculated with and without the weight decay loss and shown in separate tables to give more information.

α	CE Loss of training data	CE Loss of validation data	CE Loss of test data
0.005	0.11452	0.12664	0.12742
0.001	0.11629	0.12947	0.12541
0.0001	0.17154	0.19000	0.17071

Table 4: The CE from using Logistic Regression changing the learning rate, α. The model was trained with $\lambda = 0.1$ and the CE Loss includes the weight decay with the same λ.

As can be seen in Table 4, α being set to 0.005 and 0.001 result in very similar CE losses. However, when it is set to 0.0001 the CE loss is noticeably higher.

α	CE Loss of training data	CE Loss of validation data	CE Loss of test data
0.005	0.07837	0.09048	0.09126
0.001	0.08537	0.09855	0.09449
0.0001	0.16231	0.18077	0.16148

Table 5: The CE from using Logistic Regression changing the learning rate, α. The model was trained with $\lambda = 0.1$ but the CE Loss does not include the weight decay.

Here in Table 5, with the removal of the weight decay, the CE losses are significantly lower for the learning rates of 0.005 and 0.001. This shows that much of the CE losses in Table 4 are due to the weight decay loss as opposed to inaccuracies. However, notice that this is not the case for when the learning rate is set to 0.0001. The losses here are indeed due to the model not fully converging in the limited number of epochs, which was also seen in the curves in section 2.2.1.

2.3 Comparison to Linear Regression

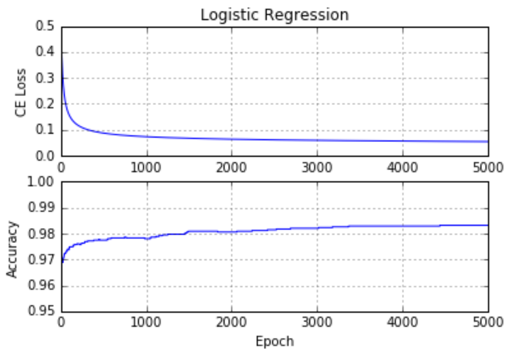

Figure 7: The CE loss and Accuracy from the training data generated using logistic regression $\alpha = 0.005$ and $\lambda = 0$.

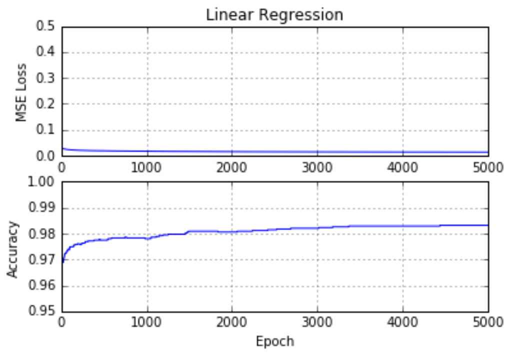

Figure 8: The MSE and Accuracy from the training data generated using linear regression $\alpha = 0.005$ and $\lambda = 0$.

In Figures 7 and 8, a comparison between linear and logistic regression can be seen. The first thing to notice is that linear regression converges to a lower MSE in comparison to logistic regression and its CE loss. However, based on the accuracy curves, the final performance of the models are actually quite similar.

This can be seen in greater detail in Table 6. Despite the MSE being lower than the CE loss, the accuracy of both types of regressions are quite similar. In fact, the Logistic regression model performs slightly better.

Regression	Accuracy of training data	Accuracy of validation data	Accuracy of test data
Logistic	0.98314	0.98	0.97931
Linear	0.97885	0.98	0.97241

Table 6: The accuracy of both regression types at classifying all three sets of data. 5000 epochs and $\alpha = 0.005$, $\lambda = 0$.

3 Batch Gradient Descent vs SGD and ADAM

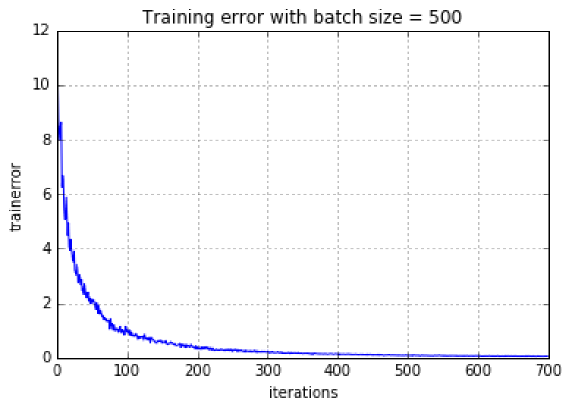

Figure 9: The M from the training data generated by the iteration using ADAM with a batch size of 500 and epoch = 700. Here $\beta1 = 0.95$, $\beta2 = 0.99$, $\epsilon = 1e - 09$.

final MSE of training data	MSE of validation data	MSE of test data
0.06749	0.09394	0.10770

Table 7: The MSE using the final optimized weight obtained using ADAM with batch size = 500 and 700 epochs. Here $\beta1 = 0.95$, $\beta2 = 0.99$, $\epsilon = 1e - 09$.

3.1 Batch size investigation

	MSE of training data	MSE of validation data	MSE of test data
B = 100	0.02293	0.04214	0.05758
B = 700	0.07079	0.13557	0.11272
B = 1750	0.15404	0.26890	0.20608

Table 8: The MSE using the final optimized weight obtained using ADAM with 700 epochs and changing the batch size, B. Here the hyper parameters for ADAM is set to $\beta1 = 0.95$, $\beta2 = 0.99$, $\epsilon = 1e - 09$.

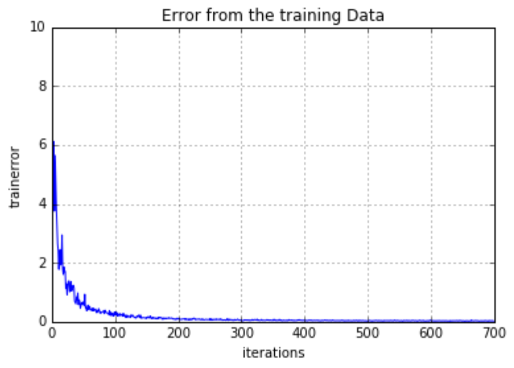

Figure 10: The MSE from the training data generated by the iteration using ADAM with a batch size of 100 and epoch = 700. Here the hyper parameters for ADAM is set to $\beta 1 = 0.95$, $\beta 2 = 0.99$, $\epsilon = 1e - 09$.

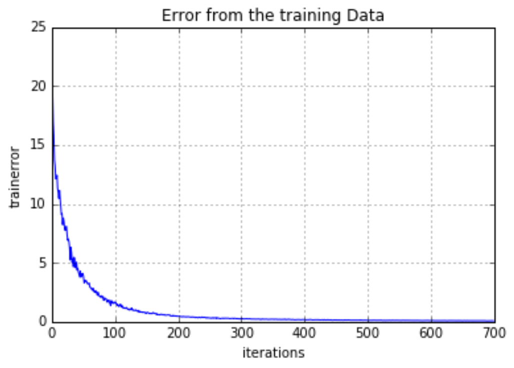

Figure 11: The MSE from the training data generated by the iteration using ADAM with a batch size of 700 and epoch = 700. Here the hyper parameters for ADAM is set to $\beta 1 = 0.95$, $\beta 2 = 0.99$, $\epsilon = 1e - 09$.

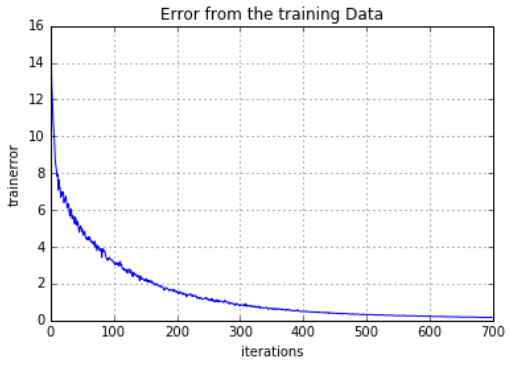

Figure 12: The MSE from the training data generated by the iteration using ADAM with a batch size of 1750 and epoch = 700. Here the hyper parameters for ADAM is set to $\beta 1 = 0.95$, $\beta 2 = 0.99$, $\epsilon = 1e - 09$.

From table 8 the final Mean squared error of using different batch sizes for ADAM is shown. This was all done with 700 epochs and one can conclude that having a smaller batch size in this case lowers the MSE for all of the data sets. From looking at the corresponding figures 10, 11 and 12 which shows the convergence of the training error for the different batch sizes the MSE for smaller batch sizes seems to converge faster but with a lot more variance and ”zig-zaging”. This seems reasonable since each gradient is taken from a smaller number of data which then does not represent the whole training set.

3.2 Hyper parameters investigation

	MSE of training data	MSE of validation data	MSE of test data
$\beta 1 = 0.95$	0.16617	0.20701	0.15927
$\beta 1 = 0.99$	0.09424	0.13010	0.12167
$\beta 2 = 0.99$	0.06485	0.12609	0.09146
$\beta 2 = 0.9999$	0.17106	0.24526	0.24778
ϵ = 1e-09	0.11658	0.14640	0.10972
ϵ = 1e-04	0.08646	0.12957	0.14502

Table 9: The final Mean Squared Error for the training data, validation data and test data changing different hyper-parameters. For each case the non-specified parameters are set to default.

The results of changing the different hyper parameters while keeping the other as default is shown in table 9 above. In ADAM $\beta 1$ scales the first moment estimates and $\beta 2$ the second moment estimates and ϵ is mainly used for numerical stability, to not divide by zero. From the table it seems that increasing $\beta 1$ will cause the MSE of all of the data sets to be smaller. This is perhaps due to the algorithm converging faster using the moment. By increasing $\beta 2$ however we MSE increases for all of the data sets. For the final parameter ϵ there is no trend across all of the data sets.

3.3 Cross Entropy Loss Investigation

final CE-loss of training data	CE-loss of validation data	CE-loss of test data
0.01955	0.04636	0.11499

Table 10: The CE-loss using the final optimized weight obtained using ADAM with batch size = 500 and 700 epochs. Here the hyper parameters are set to default and $\alpha = 0.001$.

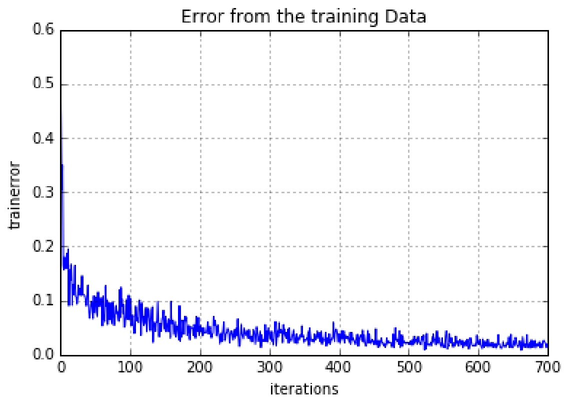

Figure 13: The CE-loss from the training data generated by the iteration using ADAM with a batch size of 500 and epoch = 700. Hyper parameters are set to default and $\alpha = 0.001$

When comparing figure 13 that shows training errors for cross entropy loss with figure 9 that shows the errors using MSE the convergence for the CE is a lot more noisy but seems to converge a lot faster. The final values shown in tabular 10 and 7 shows that also the final errors are less, however the error is not represented by the same measurement since one use CE-loss and the other MSE.

3.4 Hyper-parameter investigation for Cross Entropy Loss

	CE loss of training data	CE loss of validation data	CE loss of test data
beta1 = 0.95	0.02735	0.02815	0.12685
beta1 = 0.99	0.02040	0.02827	0.10902
beta2 = 0.99	0.01750	0.03191	0.18789
beta2 = 0.9999	0.03762	0.04234	0.12473
epsilon = 1e-09	0.02154	0.00959	0.10400
epsilon = 1e-04	0.01700	0.02613	0.14026

Table 11: The final Cross Entropy Loss for the training data, validation data and test data changing different hyper-parameters. For each case the non-specified parameters are set to default.

The result of changing the hyper parameters for Cross Entropy is shown in table 11. The trend of the CE loss decreasing for increasing $\beta1$ is not as clear as for the linear regression but it does follow the same trend. Increasing $\beta2$ does increase for CE loss as well except for the test data. Again there is no clear trend in changing ϵ. Over all the final loss seems to improve when using Logistic regression and CE instead of using Linear regression and MSE.

3.5 Comparison against Batch GD

Comparing the batch GD with SGD and ADAM we can conclude that it does decrease the final losses and also there is a clear difference in the convergence of the training error. For using SGD with mini batches we have a lot more noisy convergences but in terms of computational speed it was faster and the final error was lower overall. This is due to only optimizing with regards to part of the training set which can cause some weights to be far of optimal when comparing to the whole data set. However this is only shown in the iteration itself since after running a couple of epochs it will converge to the same optimal weights as using the full batch.