ECE1513 Winter 2019 Assignment 1

April 10, 2019

Problem 1. 1. To facilitate computing, we extended the dimensions of both W and $\mathbf{x}$ to W', $\mathbf{x}'$ such that $W' = [b, W^T]^T$ and $\mathbf{x}' = [x_0 = 1, \mathbf{x}^T]^T$ to incorporate the bias term, and define $X = [x'^{(1)T}, x'^{(2)T}, \cdots, x'^{(N)T}]$ to be the transpose of $\mathbf{x}'$.

The analytic expression for MSE can be written as:

$$\begin{aligned}\mathcal{L} = \mathcal{L}_{\mathcal{D}} + \mathcal{L}_{\mathcal{W}} &= \sum_{n=1}^{N} \frac{1}{2N} \left\| W^T \mathbf{x}^{(n)} + b - y^{(n)} \right\|_2^2 + \frac{\lambda}{2} \|W\|_2^2 \\ &= \frac{1}{2N} \sum_{n=1}^{N} \left\| W'^T \mathbf{x}'^{(n)} - y^{(n)} \right\|_2^2 + \frac{\lambda}{2} \|W\|_2^2 \\ &= \frac{1}{2N} \|XW' - \mathbf{y}\|_2^2 + \frac{\lambda}{2} \|W\|_2^2 .\end{aligned}$$

Now the analytical expression for the gradient can be found by:

$$\begin{aligned}\nabla\mathcal{L} &= \nabla \frac{1}{2N} \|XW' - \mathbf{y}\|_2^2 + \nabla \frac{\lambda}{2} \|W\|_2^2 \\ &= \frac{1}{2N} \nabla \|XW' - \mathbf{y}\|_2^2 + \frac{\lambda}{2} \nabla \|W\|_2^2 \\ &= \frac{1}{2N} \big(-2X^T\mathbf{y} + 2X^T XW' \big) + \frac{\lambda}{2}(2W) \\ &= \frac{1}{N} \big(X^T XW' - X^T\mathbf{y} \big) + \lambda W.\end{aligned}$$

In terms of function implementation, we also changed the function header for `MSE` and `gradMSE`, in order to accommodate for the extended dimensions of W' and $\mathbf{x}'$. Moreover, matrices multiplication is used instead of looping through the inputs. In the code snippet below, we use `w` to denote the weight vector W', and `X` to denote the input matrix X.

```
def MSE(w, X, y, reg):
    N = X.shape[0]
    ld = norm_sqr(np.matmul(X, w) - y) / 2 / N
    # Use only original w for regularization
    lw = norm_sqr(w[1:]) * reg / 2
    return ld + lw
```

```
def gradMSE(w, X, y, reg):
    N = X.shape[0]
    return (
        (np.matmul(np.matmul(X.T, X), w) - np.matmul(X.T, y)) / N
        + reg * np.concatenate([[0], w[1:]])
    )
```

2. The implementation of the batch Gradient Descent algorithm is carried out in the following functions:

- `grad_descent_step`: performs one step of the Gradient Descent algorithm. It calculates the loss and the gradient in the current step using functions MSE and gradMSE from above and updates the weight.
- `full_grad_descent`: performs the full Gradient Descent algorithm. It calls grad_descent_step function iteratively until the required number of epochs or the desired error tolerance is reached. It returns the optimized weight and also stores the training, validation and test losses/accuracies.

Refer to the source code submitted separately for detailed implementation of the functions.

3. Figure 1 plots the training, validation, and test losses of the Gradient Descent algorithm under settings $\lambda = 0$, epochs=5000, $\alpha = [0.005, 0.001, 0.0001]$. As shown in the figure, greater α results in faster convergence.

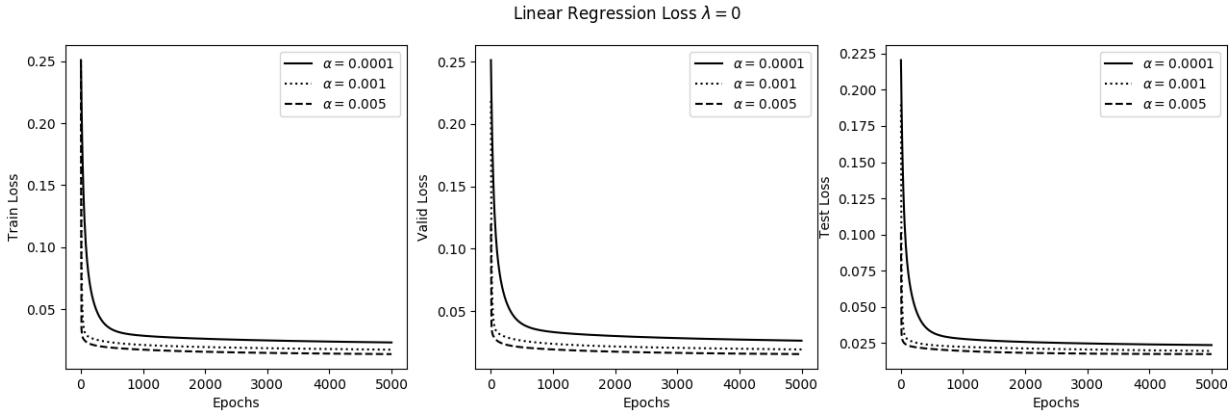

Figure 1: Training loss (top left), Validation loss (top right), and Test loss (bottom left) under the settings $\lambda = 0$, epochs = 5000 and $\alpha = [0.005, 0.001, 0.0001]$.

Table 1 summarizes the final accuracy and training time with different α values. After 5000 epochs, greater α tends to achieve better accuracy on all training, validation, and test data sets. This can be considered as the consequence of faster convergence. On the other hand, the training time under different α settings do not appear to differ significantly, which is reasonable since all settings have to perform the same number of 5000 epochs.

α	training accu.	validation accu.	test accu.	training time
0.005	97.89%	98.00%	97.24%	195.9s
0.001	96.31%	97.00%	95.86%	197.6s
0.0001	94.74%	93.00%	95.17%	203.1s

Table 1: Final (epochs=5000) classification accuracy and training time with different α settings.

4. To investigate the effect of the regularization parameter λ on the performance of the algorithm, the training, validation, and test losses under settings $\lambda = [0.001, 0.1, 0.5]$, $\alpha = 0.005$ and epochs=10000 are plotted in Figure 2 on the next page. As shown in the figure, smaller λ values appear to achieve lower training, validation and test losses. In the cases where $\lambda = 0.5$ and $\lambda = 0.1$, the training terminates after 4786 and 5145 epochs, since the error tolerence has been reached. It's worth noting that when $\lambda = 0.001$, even though the training and validation losses continue to decrease, the test loss start to increase after around 5610 epochs, indicating the sign of overfitting.

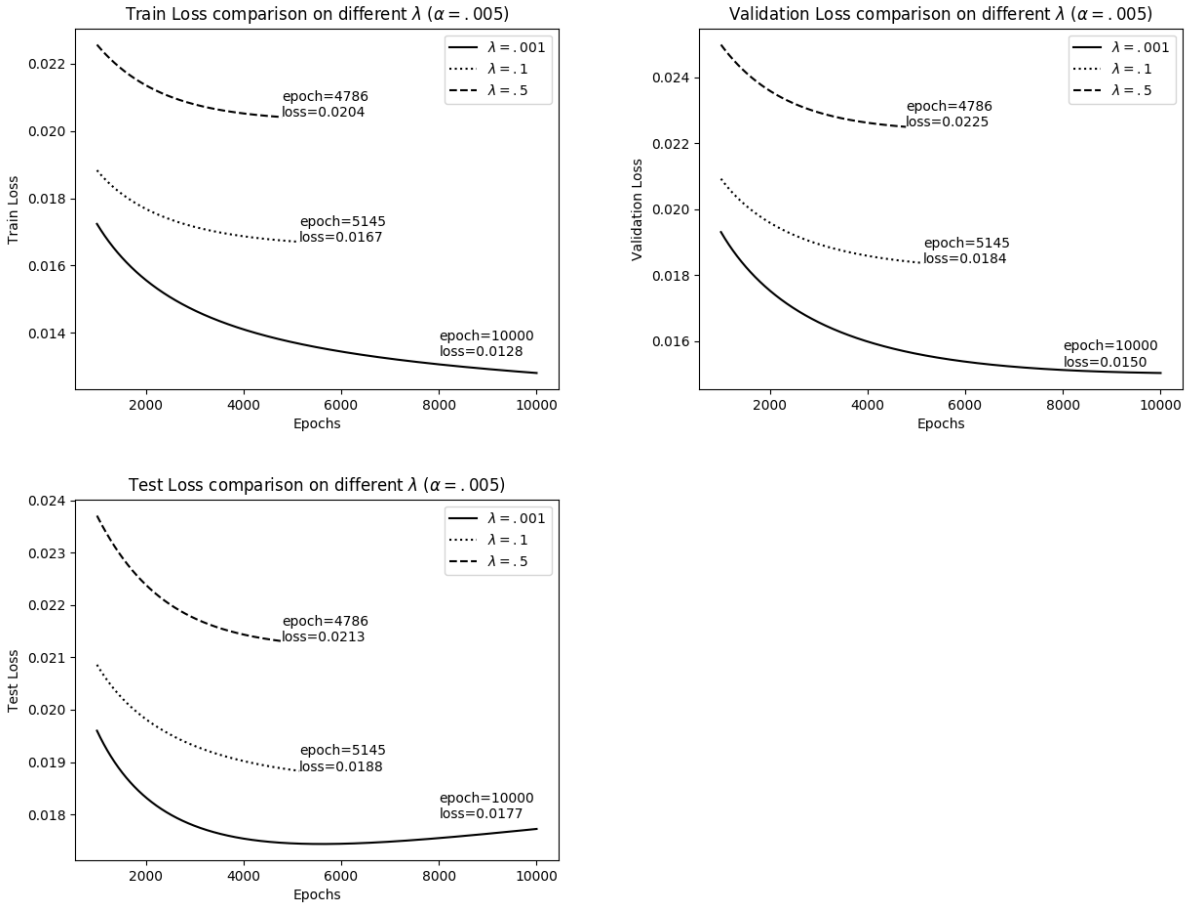

Figure 2: Training loss (top left), Validation loss (top right), and Test loss (bottom left) under the settings $\lambda = [0.001, 0.1, 0.5]$, $\alpha = 0.005$ and epochs=10000.

Table 2 summarizes the final training, validation and test accuracy using different λ values. Choosing smaller λ values appear to enhance final training, validation and test accuracy performance in this case. However, since the error tolerance is reached after 4786 epochs for $\lambda = 0.5$ and 5145 epochs for $\lambda = 0.1$, the training terminates much sooner than in the case where $\lambda = 0.001$ and all 10000 epochs need to be completed. As a result, training time is greatly reduced when larger λ values are used.

λ	epochs	training accu.	validation accu.	test accu.	training time
0.001	10000	98.29%	98.00%	97.24%	539.37s
0.1	5145	97.6%	97.00%	96.55%	269.72s
0.5	4786	97.11%	97.00%	96.55%	263.09s

Table 2: Final classification accuracy and training time under different λ settings. Note when $\lambda = 0.1$ and $\lambda = 0.5$, the iteration ends at 5145 and 4786 epochs respectively due to reaching the error tolerance 1×10^{-7}.

The rationale behind tuning λ using the validation set is usually to prevent overfitting. As the model is trained, it may fit the training set increasingly well, but at some point start to lose its generalizability over broader data sets. The regularization parameter λ is included into the cost function to prevent overfitting and the proper value of λ is selected from fitting the model over the validation set.

5. The following function is written to find the optimum weights using the normal equation for the derivative of the mean square error.

```
def linear_regression_normal_equation(X, y, reg):

    """Use analytical solution of the means square error to find optimum
       weights
    W* = (X^T * X + \lambda * I)^-1 * X^T * Y
    """

    d = X.shape[1]
    return np.matmul(
        np.matmul(
            np.linalg.inv(
                np.matmul(X.T, X) + reg * np.eye(d)
            ),
            X.T
        ),
        y
    )
```

Table 3 summarizes the final error, accuracy and training time of the normal equation algorithm when λ is set to zero. When compared to the results of the Batch Gradient Descent algorithm from part 3, it is observed that the normal equation method obtains lower training error and higher training accuracy. This is as expected as the normal equation is the

analytical solution to the mean squared error loss function. The validation and testing errors are comparable for both algorithms. Since the optimum weights are calculated directly in the normal equation method, the algorithm terminates after 1 epoch, resulting in a much reduced computation time.

training error	validation error	test error	training time
0.009352	0.02378	0.02849	0.335s
training accuracy	validation accuracy	test accuracy	
99.37%	96%	94.48%	

Table 3: The final error, accuracy and computation time information when the normal equation of the least squares formula is used to calculate the optimum weights. λ is set to zero for comparison with Batch Gradient Descent from part 3.

Problem 2. 1. In order to solve for the gradient of the cross entropy error, we first let $f = -y^{(n)} \log \hat{y}\big(\mathbf{x}^{(n)}\big) - \big(1 - y^{(n)}\big) \log \big(1 - \hat{y}\big(\mathbf{x}^{(\mathbf{n})}\big)\big)$ and $z = W^T\mathbf{x} + b$. We also introduce the same notation W' and $\mathbf{x}'$ to extend dimensions of W and x and incorporate the bias term as in Problem 1.1. Then we have

$$\begin{aligned}
f &= -y^{(n)} \log \sigma(z) - (1 - y^{(n)}) \log(1 - \sigma(z)) \\
&= -y^{(n)} \log \frac{1}{1 + e^{-z}} - (1 - y^{(n)}) \log \frac{e^{-z}}{1 + e^{-z}} \\
&= y^{(n)} \log(1 + e^{-z}) - (1 - y^{(n)}) \log e^{-z} + (1 - y^{(n)}) \log(1 + e^{-z}) \\
&= (1 - y^{(n)})z + \log(1 + e^{-z}) \\
&= z - zy^{(n)} + \log(1 + e^{-z}) \\
&= \log e^{z} - zy^{(n)} + \log(1 + e^{-z}) \\
&= -zy^{(n)} + \log(1 + e^{z}).
\end{aligned}$$

And the gradient is

$$\begin{aligned}
\nabla f &= \nabla(-zy^{(n)}) + \nabla \log(1 + e^{z}) \\
&= -y^{(n)} \nabla z + \nabla \log(1 + e^{z}) \\
&= -y^{(n)} \nabla z + \frac{1}{1 + e^{z}} \nabla(1 + e^{z}) \\
&= -y^{(n)} \nabla z + \frac{e^{z}}{1 + e^{z}} \nabla z \\
&= \big(\hat{y}(\mathbf{x}^{(n)}) - y^{(n)}\big) \nabla z.
\end{aligned}$$

Since $\nabla z = \nabla W'^{T} \mathbf{x}'^{(n)} = \mathbf{x}'^{(n)}$, we have $\nabla f = \big(\hat{y}(\mathbf{x}'^{(n)}) - y^{(n)}\big)\mathbf{x}'^{(n)}$.

Finally, the gradient of the cross entropy loss is found to be

$$\begin{aligned}\nabla\mathcal{L} &= \nabla\sum_{n=1}^{N}\frac{1}{N}f + \nabla\frac{\lambda}{2}\|W\|_2^2 \\ &= \frac{1}{N}\sum_{n=1}^{N}\nabla f + \lambda W \\ &= \frac{1}{N}\sum_{n=1}^{N}\big(\hat{y}(\mathbf{x}'^{(n)}) - y^{(n)}\big)\mathbf{x}'^{(n)} + \lambda W.\end{aligned}$$

The cross entropy loss and its gradient are implemented through the following functions:

```
def crossEntropyLoss(w, X, y, reg):
    N, d = X.shape
    total = 0
    for i in range(N):
        z = np.matmul(w, X[i])
        total += -z * y[i] + np.log(1 + np.exp(z))
    ld = total / N
    # Use only original w for regularization
    lw = norm_sqr(w[1:]) * reg / 2
    return ld + lw
```

```
def gradCE(w, X, y, reg):
    """Gradient for cross entropy loss"""
    N, d = X.shape
    grad = np.zeros(d)
    for i in range(N):
        z = np.matmul(w, X[i])
        grad = grad + (sigmoid(z) - y[i]) * X[i]
    return grad / N + reg * np.concatenate([[0], w[1:]])
```

2. The functions `grad_descent_step` and `full_grad_descent` from part 1.2 are modified by including a flag to specify which type of loss/gradient to use for the classifier. Refer to the source code for more details of the functions.

Figure 3 plots the loss and accuracy curves of the logistic regression model when $\lambda = 0.1$, epochs $= 5000$ and $\alpha = 0.005$.

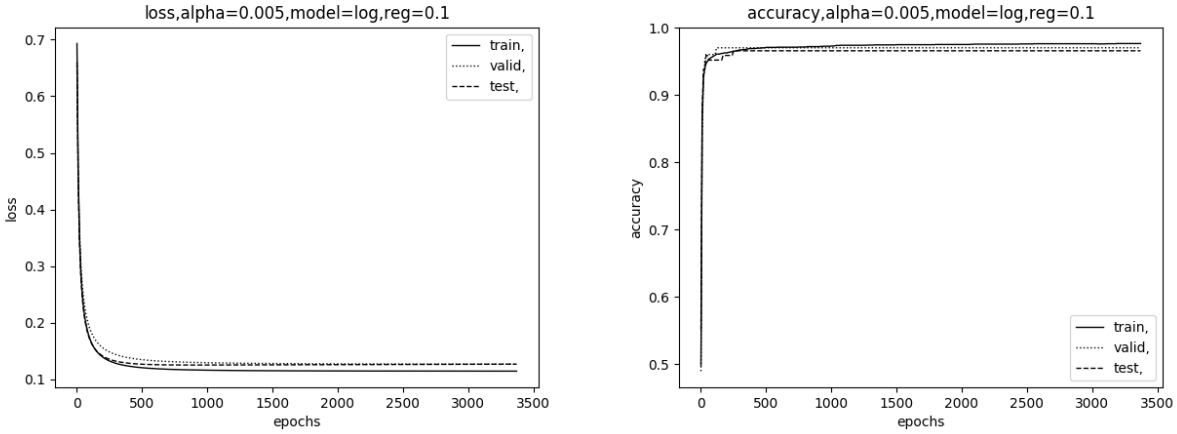

Figure 3: Loss (left), Accuracy (right) curves of the logistic regression model using cross entropy loss under settings $\lambda = 0.1$, epochs $= 5000$ and $\alpha = 0.005$.

3. Figure 4 plots the training loss and accuracy behaviour for the logistic regression model using cross entropy loss and the linear regression model using MSE loss, under the settings $\lambda = 0$, epochs $= 5000$ and $\alpha = 0.005$. It's observed from the graph that the linear regression model using MSE loss appear to converge faster. Nevertheless, the logistic regression model using cross entropy loss manage to gain higher training accuracy.

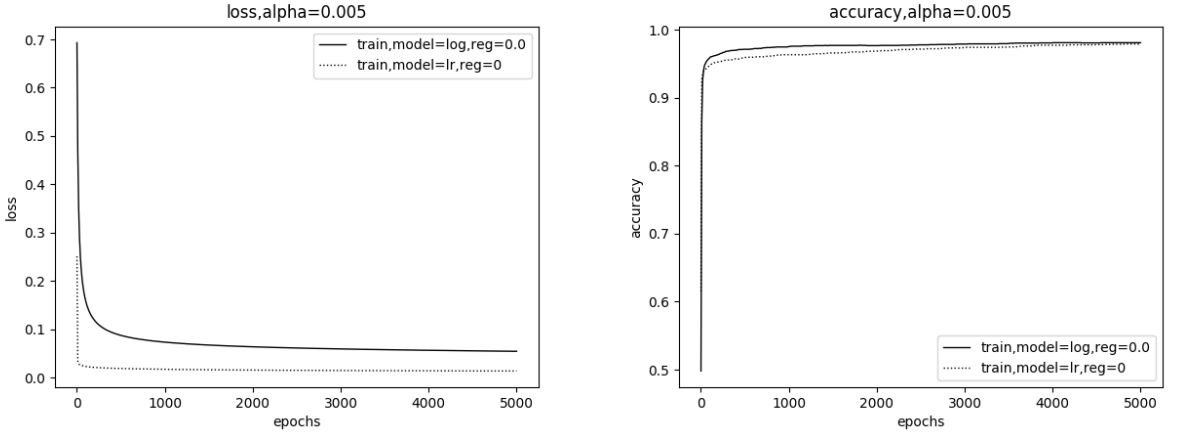

Figure 4: Traing loss (left), accuracy (right) comparison of the logistic regression model using cross entropy loss (solid line) and the linear regression model with MSE loss (dashed line), under the settings $\lambda = 0$, epochs $= 5000$ and $\alpha = 0.005$.

Problem 3. 1. For simplicity we created a separate file named `a1-tf.py` for TensorFlow implementions. This file contains SGD algorithm with MSE and Cross Entropy loss, as well as the `GradientDescentOptimizer` and `AdamOptimizer`. A code snippet is given below. For more details, refer to the source code.

```
def buildGraph(d, optimizer_type, loss_type, params):
    X = tf.placeholder(tf.float32, shape=(None, d), name='X')
    y = tf.placeholder(tf.float32, shape=(None,), name='y')
    w = tf.Variable(tf.truncated_normal((d,), stddev=0.5), name='w')
    # w = tf.Variable(tf.zeros(d,), tf.float32, name='w')
    reg = tf.placeholder(tf.float32, shape=(), name='reg')

    f = tf.matmul(X, tf.reshape(w, shape=(-1, 1)))
    y_ = tf.reshape(f, shape=(-1,))

    if loss_type == 'mse':
        # loss = tf.losses.mean_squared_error(y, y_)
        loss = tf.reduce_mean(tf.square(y - y_))
    elif loss_type == 'ce':
        loss = tf.losses.sigmoid_cross_entropy(y, y_)
    loss = loss + reg * tf.nn.l2_loss(w[1:])

    if optimizer == 'gd':
        opt_op = tf.train.GradientDescentOptimizer(
            learning_rate=params['alpha'],
        ).minimize(loss)
    else:
        opt_op = tf.train.AdamOptimizer(
            learning_rate=params['alpha'],
            beta1=params['beta1'],
            beta2=params['beta2'],
            epsilon=params['epsilon'],
        ).minimize(loss)

    predict = (tf.sign(y_ - 0.5) + 1) / 2
    accuracy = tf.reduce_mean(tf.cast(tf.equal(y, predict), tf.float32))

    return X, y, y_, w, reg, loss, predict, accuracy, opt_op
```

2. The Stochastic Gradient Descent algorithm is implemented to minimize the MSE, under the settings of minibatch size=500, epochs=700, $\alpha = 0.001$ and $\lambda = 0$. The resulting loss and accuracy curves are presented in figure 5. As seen in the figure, the loss and accuracy curves of the SGD are not as smooth as in the Batch Gradient Algorithm. The overall trend, however, is still to reduce losses and improve accuracy over time. The reason is that each batch update does not reflect the overall gradient, and the accumulation of many updates is what is expected to approximate the effects of overall gradient.

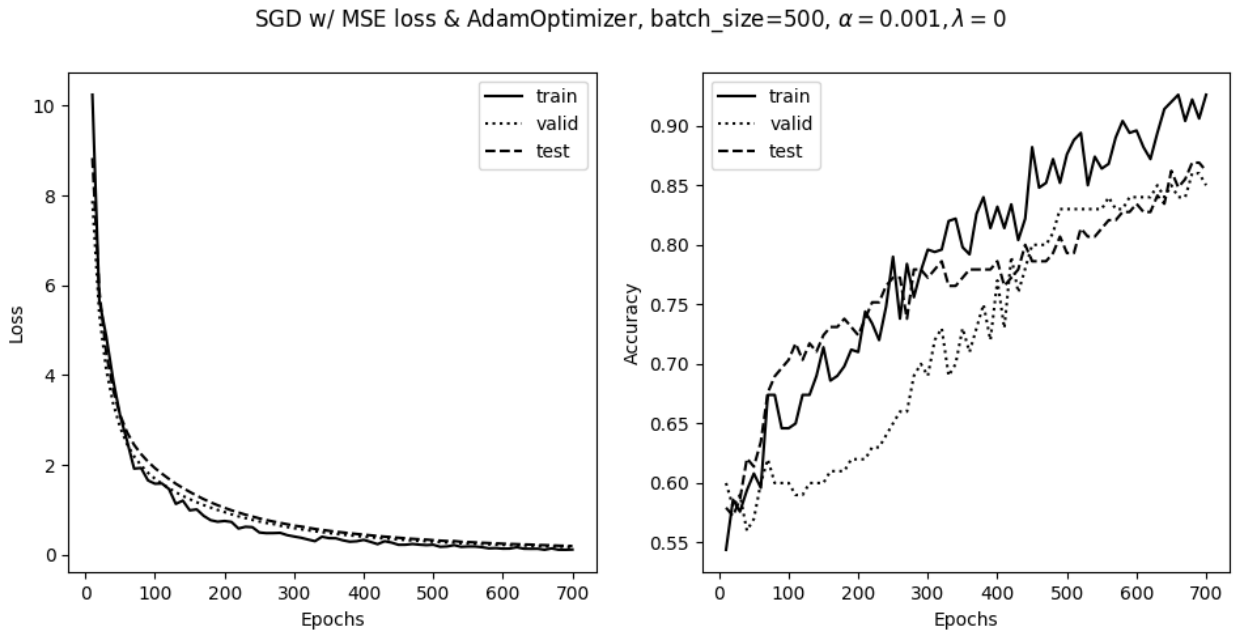

Figure 5: SGD algorithm is implemented to minimize the MSE using minibatch size=500 , epochs =700, $\alpha = 0.001$ and $\lambda = 0$. Training, validation and test loss curves are depicted on the left, with the corresponding accuracy curves on the right.

3. To investigate the effects of batch size on the behaviour of the SGD algorithm using Adam, three different batch sizes, namely 100, 700 and 1750 are utilized to plot the loss and accuracy curves in figure 6. Smaller batch size is found to reduce loss and improve accuracy in this case. The rationale for this is that smaller batch size allows for a greater number of updates in each epoch, thus improves the overall performance of the classification.

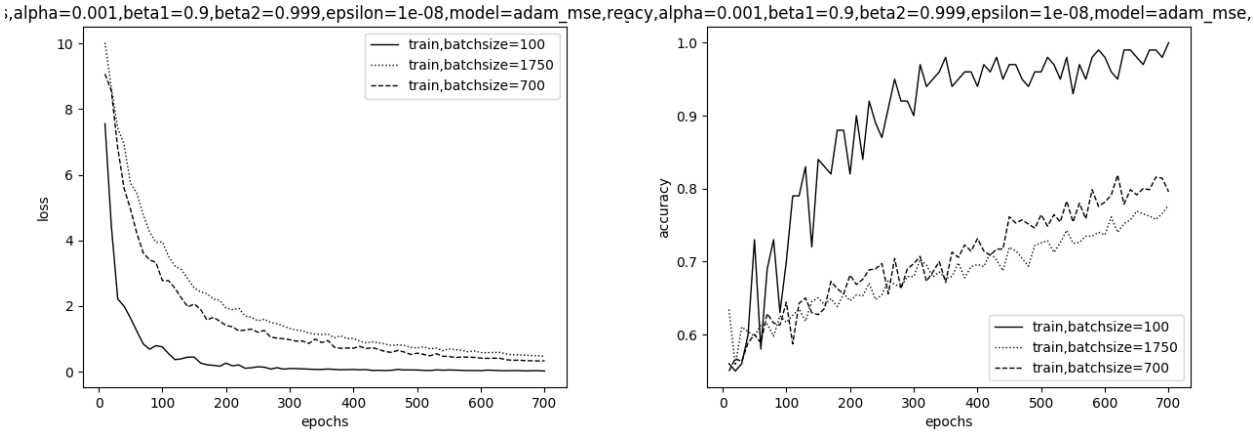

Figure 6: Three different batch sizes 100, 700, 1750 are used in the SGD algorithm to minimize the MSE. The other settings are epochs =700, $\alpha = 0.001$ and $\lambda = 0$. Training loss (left) and accuracy (right) curves are plotted.

4. Next we experiment with different values for Adam hyperparameters to investigate their effects on the final training, validation and test accuracy. The minibatch size is set to 500, $\alpha = 0.001$ and epochs=700.

Tensorflow uses the algorithm proposed by Kingma and Ba [1]. The update rule of the algorithm is listed in Algorithm 1.

Algorithm 1 Adam

θ_0: initial parameter
$m_0 \leftarrow 0$ ▷ First moment vector
$v_0 \leftarrow 0$ ▷ Second moment vector
$t \leftarrow 0$
while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
    $\alpha_t \leftarrow \alpha \cdot \sqrt{1 - \beta_2^t}/(1 - \beta_1^t)$
    $\theta_t \leftarrow \theta_{t-1} - \alpha_t \cdot m_t/(\sqrt{v_t} + \epsilon)$

(a) $\beta_1 = (0.95, 0.99)$: β_1 is the exponential decay rate for the first moment estimate. From the update rule $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$, we know that β_1 controls the weight of first moment (setting $\beta_1 = 0$ results in regular SGD without moment), and creates a "smoother" gradient than the raw gradient g_t.

Table 4 lists the accuracy results from $\beta_1 = \{0.95, 0.99\}$ and we can see that there is no significant difference on the final accuracy. Figure 7 shows the comparison of training loss and accuracy. From the plot we can see that greater $\beta_1 = 0.99$ has smaller gradient component which results in slightly slower converge.

β_1	training	valid	test
0.95	92.00%	88.00%	88.28%
0.99	91.60%	87.00%	87.59%

Table 4: Comparison of accuracy on $\beta_1 = \{0.95, 0.99\}$. Other hyperparameters are $\beta_2 = 0.999, \epsilon = 10^{-8}$.

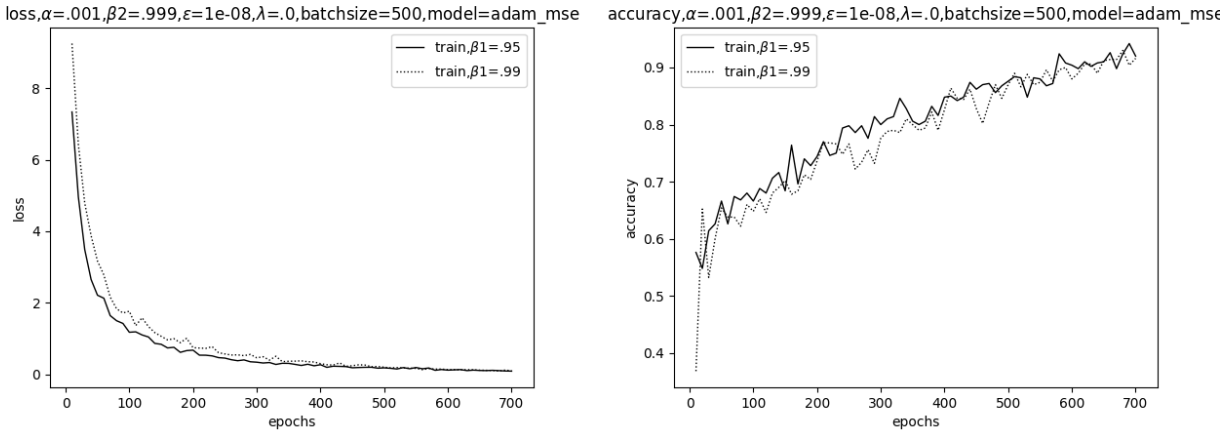

Figure 7: Training loss and accuracy for $\beta 1 = 0.95$ and 0.99.
β_2 and ϵ are set to default Tensorflow initialization, batch size=500, $\alpha = 0.001$ and epochs=700.

(b) $\beta_2 = (0.99, 0.9999)$: β_2 is the exponential decay rate for the second moment. From $\alpha_t \leftarrow \alpha \cdot \sqrt{1-\beta_2^t}/(1-\beta_1^t)$ we know that increasing β_2 will effectively decrease the learning rate.

Table 5 lists the accuracy on $\beta_2 = \{0.99, 0.9999\}$ and we can see that $\beta_2 = 0.99$ has a higher final accuracy, which indicates $\beta_2 = 0.99$ converges faster than $\beta_2 = 0.9999$. Figure 8 shows the comparison of training loss and accuracy. From the plot we can see that greater $\beta_2 = 0.9999$ results in slower converge.

β_2	training	valid	test
0.99	94.20%	90.00%	91.03%
0.9999	86.00%	87.00%	80.68%

Table 5: Comparison of accuracy on $\beta_2 = \{0.99, 0.9999\}$. Other hyperparameters are $\beta_1 = 0.9, \epsilon = 10^{-8}$.

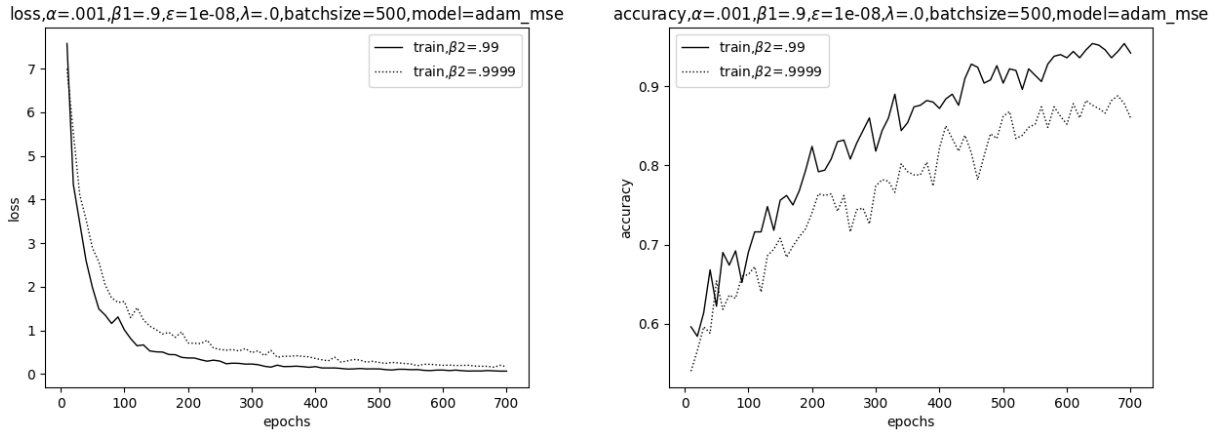

Figure 8: Training loss and accuracy for $\beta 2 = 0.99$ and 0.9999.
$\beta 1$ and ϵ are set to default Tensorflow initialization, batch size=500, $\alpha = 0.001$ and epochs=700.

(c) $\epsilon = (1e^{-9}, 1e^{-4})$: ϵ is a small number to avoid zero. However a very small ϵ results in large weight updates and could potentially result overshooting.

Table 6 lists the accuracy results on $\epsilon = \{10^{-9}, 0.0001\}$ and we can see that $\epsilon = 0.0001$ has a higher final accuracy.

ϵ	training	valid	test
10^{-9}	89.00%	89.00%	82.07%
0.0001	92.20%	89.00%	87.59%

Table 6: Comparison of accuracy on $\epsilon = \{10^{-9}, 0.0001\}$. Other hyperparameters are $\beta_1 = 0.9, \beta_2 = 0.999$.

5. We rerun 3.2 - 3.4 with cross entropy loss function. The final loss and accuracy for different batch size and hyperparameter setting is listed in Table 7.

Generally, cross entropy loss meets the analysis we did in 3.2 - 3.4. Overall, however, cross entropy has less loss and higher accuracy than MSE loss at epoch=700, indicating that cross entropy loss converges faster than MSE loss. We may conclude that for binary classification problem, logistic regression with cross entropy loss is a "more suitable" model than linear regression.

Optimizer Type and Parameter Value	MSE loss	CE loss	MSE accuracy	CE accuracy
Adam, batch=100	0.022417	0.001765	1.000000	1.000000
Adam, batch=700	0.334083	0.028498	0.795714	0.990000
Adam, batch=1750	0.471597	0.044727	0.777143	0.982857
Adam, beta1=0.95	0.094182	0.025064	0.920000	0.990000
Adam, beta1=0.99	0.104681	0.025137	0.916000	0.990000
Adam, beta2=0.99	0.063865	0.011852	0.942000	0.998000
Adam, beta2=0.9999	0.163707	0.035279	0.860000	0.982000
Adam, epsilon=1e-9	0.145734	0.015310	0.890000	0.994000
Adam, epsilon=0.0001	0.102104	0.024191	0.922000	0.990000

Table 7: Comparison of MSE loss and Cross Entropy loss under different settings, at epoch=700. Data from training loss and training accuracy.

6. Figure 9 plots the loss and accuracy curves of the SGD and batch gradient descent algorithm for comparison.

SGD does not have a smooth loss curve as Batch GD does. The reason is that each step in SGD considers only a small batch of data for optimizing loss, thus does not guarantee global loss to decrease.

However, SGD converges faster than Batch GD. Table 8 shows that at epoch=10, SGD converges to loss = 0.0427 while Batch GD still has loss = 0.0924. However at epoch 100 and epoch 500, since SGD cannot guarantee monotonic decreasing of loss, it may have greater loss than Batch GD.

Epochs	SGD with Adam	Batch GD
10	0.0427	0.0924
100	0.0296	0.0285
500	0.0261	0.0231

Table 8: Training loss comparison of Adam SGD vs. Batch GD.

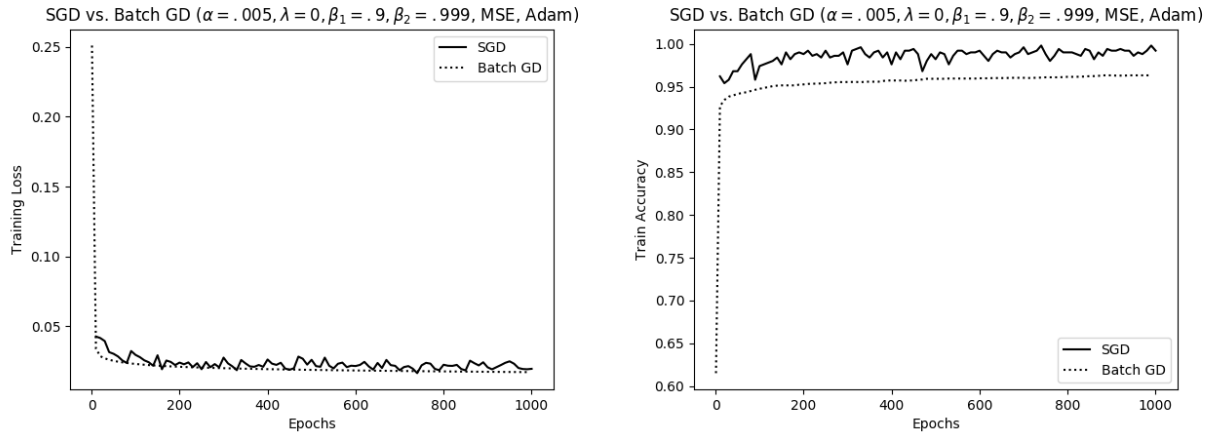

Figure 9: Comparison of SGD with Adam optimizer vs. Batch GD on training loss (left) and training accuracy (right) under the settings $\lambda = 0$ and $\alpha = 0.005$.
For comparison, both SGD and Batch GD use zero vector as initial $\mathbf{w}$ and b value.

References

[1] Kingma and Ba, *Adam: A Method for Stochastic Optimization*. https://arxiv.org/abs/1412.6980