ECE421 – Introduction to Machine Learning

Assignment 1

Linear and Logistic Regression

Hard Copy Due: Friday, Feb. 7 @ 2:00 PM, at BA3128
Code Submission Due: Friday, Feb. 7 @ 2:00 PM, on Quercus

General Notes:

- Attach this cover page to your hard copy submission
- Please post assignment related questions on Piazza.

Please check section to which you would like the assignment returned.

Tutorial Sections:

☐ Tutorial 1: Thursdays 3-5pm (SF2202)
☐ Tutorial 2: Thursdays 3-5pm (GB304)
☐ Tutorial 3: Tuesdays 10-12 (SF2202)
☐ Tutorial 4: Fridays 9-11 (BA1230)

Group Members	
Name	Student ID

ECE 421 Assignment 1

Calvin Ma 1003803805
Contribution: 50%
Jerry He 1003979180
Contribution: 50%

February 7, 2020

Calvin Ma 1003803805
Jerry He 1003979180

Assignment 1

ECE 421
February 7, 2020

1 Linear Regression

1.1 Loss Function and Gradient

The Mean Squared Error (MSE) loss function was given as:

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

where an underlined symbol denotes a vector quantity. The Python implementation of the MSE loss function is shown below:

```
def MSE(W, b, x, y, reg):
    e_in = np.matmul(x, W) + b - y
    L_D = np.sum(np.square(np.linalg.norm(e_in))) / x.shape[0]
    L_W = np.square(np.linalg.norm(W)) * reg / 2
    return L_D + L_W
```

Taking the partial derivative of (1) with respect to w_i for some $i \in [1, d]$, we obtain:

$$\frac{\partial \mathcal{L}}{\partial w_i} = \sum_{n=1}^{N} \frac{2}{N} \left\| \underline{W}^T \underline{\mathbf{x}}^{(n)} + b - y^{(n)} \right\|_2 x_i^{(n)} + 2\lambda w_i$$

where the gradient of $\mathcal{L}$ with respect to W is simply:

$$\underline{\nabla}_{\underline{W}} \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \frac{\partial \mathcal{L}}{\partial w_2} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial w_d} \end{bmatrix} \tag{2}$$

Taking the partial derivative of (1) with respect to b, we obtain:

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_{n=1}^{N} \frac{2}{N} \left\| \underline{W}^T \underline{\mathbf{x}}^{(n)} + b - y^{(n)} \right\|_2$$

The Python implementation of the MSE loss function gradient is shown below:

```
def gradMSE(W, b, x, y, reg):
    e_in = np.matmul(x, W) + b - y
    dL_D_dw = 2 * (np.matmul(np.transpose(x), e_in)) / x.shape[0]
    dL_W_dw = 2 * reg * W
    dL_D_db = 2 * np.sum(e_in) / x.shape[0]
    return dL_D_dw + dL_W_dw, dL_D_db
```

1.2 Gradient Descent Implementation

The gradients of the loss function as shown in (2) gives the directions of steepest ascent. Thus, the negative gradients gives the directions of steepest descent. Multiplying the negative gradient by the learning rate α, the following update rules are obtained:

$$\underline{W}_{k+1} \leftarrow \underline{W}_k - \alpha \nabla_{\underline{W}_k} \mathcal{L}$$
$$b_{k+1} \leftarrow b_k - \alpha \frac{\partial \mathcal{L}}{\partial b_k}$$

The larger the learning rate α, the larger the step the model takes every epoch.

The Python implementation of the gradient descent algorithm is shown below:

```
def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol=10**(-7)):
    for i in range(epochs):
        dW, db = gradMSE(W, b, x, y, reg)
        W = W - alpha * dW
        b = b - alpha * db
        if (np.linalg.norm(alpha * dW) <= error_tol):
            break
        loss.append(MSE(W, b, x, y, reg))
    return W, b
```

1.3 Tuning the Learning Rate

The weights and bias were randomly initialized using NumPy. Figure 1 plots the training, validation, and test losses and accuracies for step sizes of $\alpha = 0.005$, 0.001, and 0.0001. The largest step size of $\alpha = 0.005$ minimized the loss function the fastest and achieved the highest accuracy across all three data sets, as confirmed by the results in Table 1. The smaller step sizes converged more slowly and were unable to achieve a high accuracy within 5000 epochs. In general, smaller step sizes take longer to converge and thus require more computation time and power. However, large step sizes may overshoot the minimum and in extreme cases, may diverge.

Table 1: Final accuracies of Linear Regression for learning rates of $\alpha = 0.005$, 0.001, and 0.0001.

	$\alpha = 0.005$	$\alpha = 0.001$	$\alpha = 0.0001$
Training data accuracy	0.7917	0.6571	0.5846
Validation data accuracy	0.76	0.69	0.59
Test data accuracy	0.7379	0.6966	0.6138

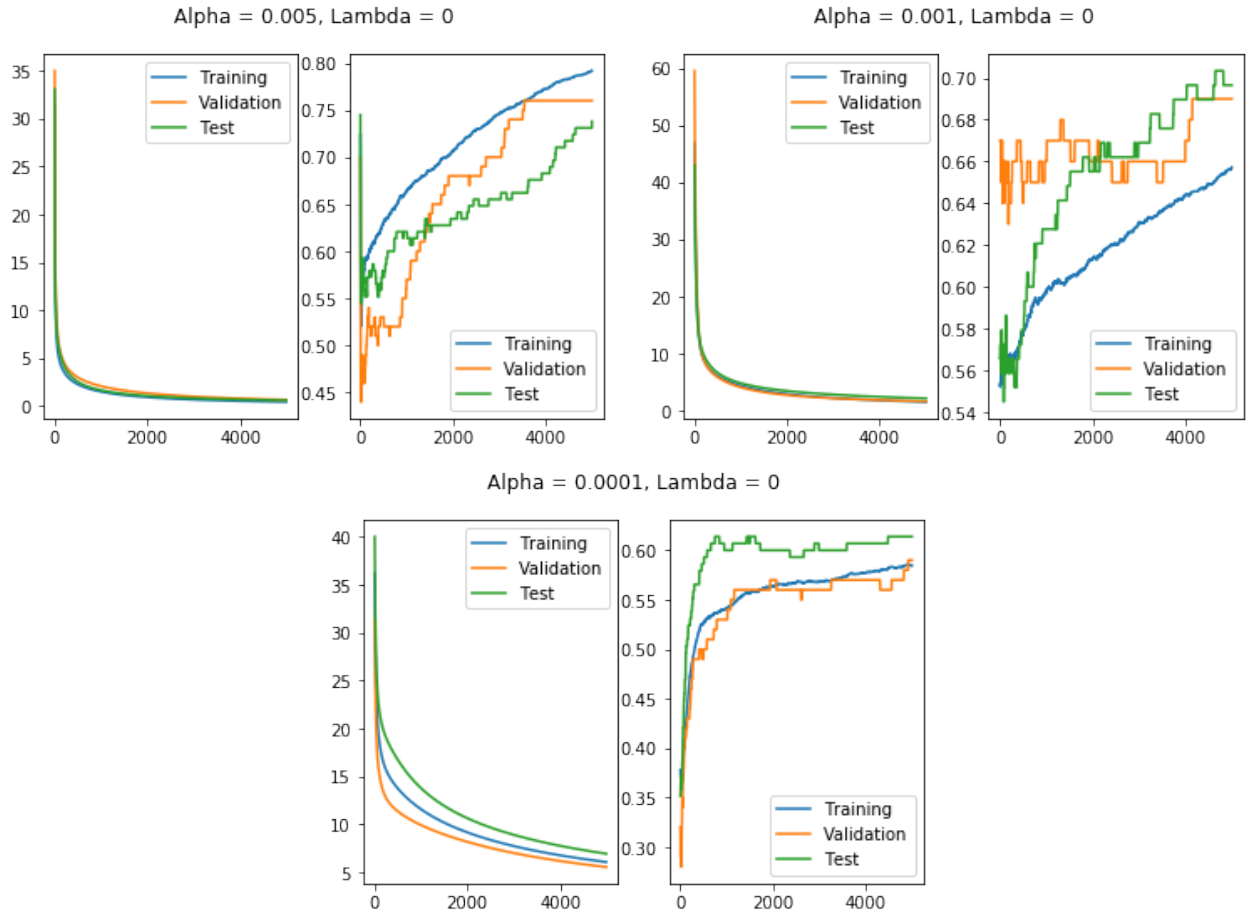

Figure 1: Linear Regression loss and accuracy for learning rates of $\alpha = 0.005$, 0.001, and 0.0001.

1.4 Generalization

For $\alpha = 0.005$ in 1, the model achieved a higher accuracy on the training data set than the validation and test data sets, which is a sign of over-fitting. The regularization term λ introduces weight decay which helps combat over-fitting. When λ is small, as in the $\lambda = 0.001$ case, there is still evidence of over-fitting, as the training accuracy is still higher than validation and test accuracy (Figure 2). When *lambda* is increased to 0.1, the accuracy of the model on all three data sets is much higher and nearly identical (Table 2). For $\lambda = 0.5$, the accuracies remain similar to when $\lambda = 0.1$, but the loss function converges much quicker with no signs of over-fitting. In Figure 2, we can see that the loss function converges in around 500 epochs for $\lambda = 0.5$, versus ~2000 epochs for $\lambda = 0.1$. Therefore, $\lambda = 0.5$ is the best choice of regularization parameter out of the three.

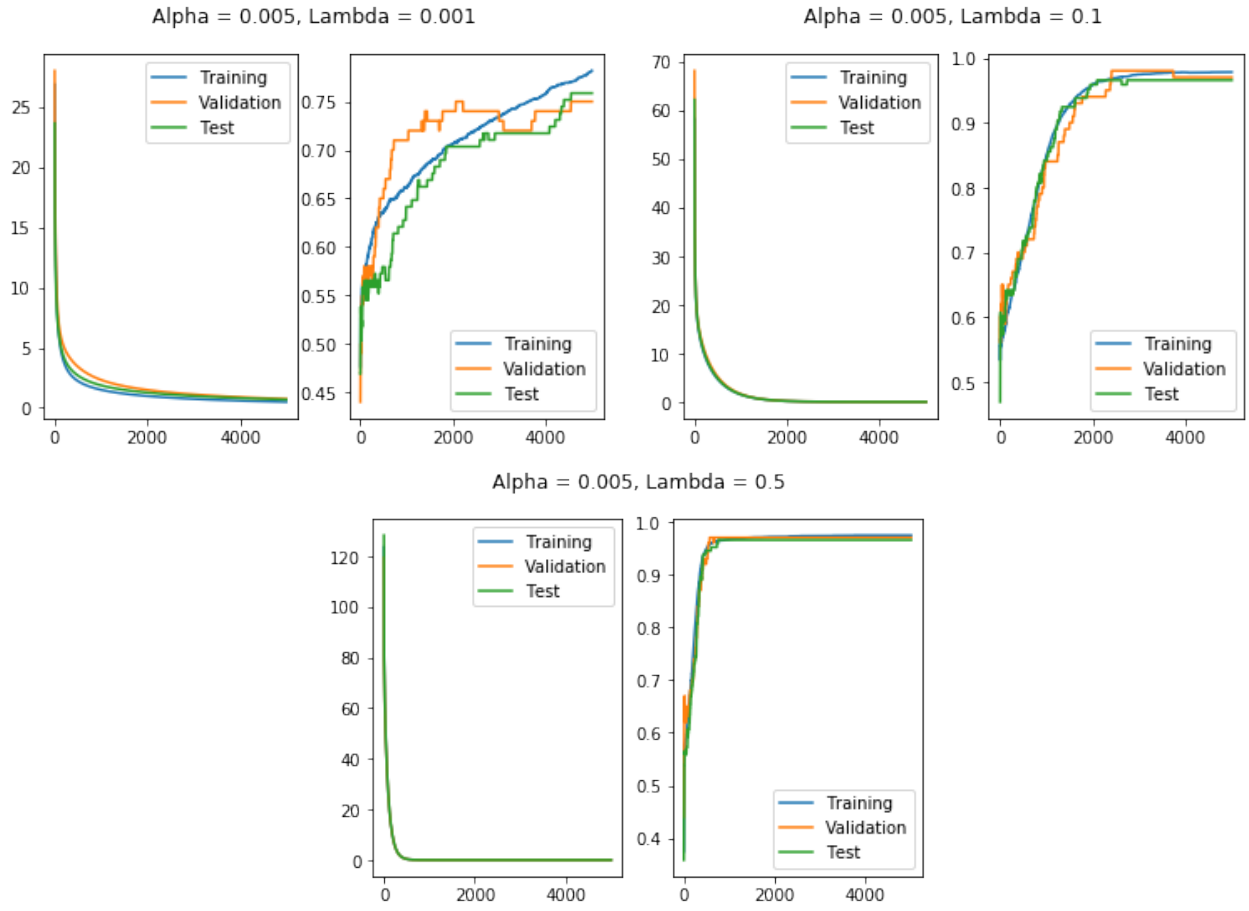

Figure 2: Linear Regression loss and accuracy for regularization parameter values of $\lambda = 0.001$, 0.1, and 0.5.

Table 2: Final accuracies of Linear Regression regularization parameter values of $\lambda = 0.001$, 0.1, and 0.5.

	$\lambda = 0.001$	$\lambda = 0.1$	$\lambda = 0.5$
Training data accuracy	0.7817	0.978	0.9742
Validation data accuracy	0.75	0.97	0.97
Test data accuracy	0.7586	0.9655	0.9655

1.5 Comparing Batch GD with normal equation

Using the original matrices

$$X = \begin{bmatrix} x_{11} & \dots & x_{1d} \\ \vdots & \ddots & \vdots \\ x_{N1} & \dots & x_{Nd} \end{bmatrix} \text{ and } W = \begin{bmatrix} w_1 \\ \vdots \\ w_d \end{bmatrix}$$

the augmented matrices $\widetilde{X}$ and $\widetilde{W}$ are constructed as follows:

$$\widetilde{X} = \begin{bmatrix} 1 & x_{11} & \dots & x_{1d} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N1} & \dots & x_{Nd} \end{bmatrix} \text{ and } \widetilde{W} = \begin{bmatrix} b \\ w_1 \\ \vdots \\ w_d \end{bmatrix}$$

where b is the bias.

Using the augmented matrices, the normal equation of the least squares formula can be expressed as

$$W^* = (X^T X)^{-1} X^T Y$$

where W^* is the optimized weight vector and Y is the observation vector.

The following script implements the normal equation of the least squares formula using numpy:

```
onesCol = np.ones([x.shape[0], 1])
xTilde = np.concatenate((onesCol, x), axis=1)
wStar = np.matmul(np.linalg.inv(np.matmul(np.transpose(xTilde), xTilde)),
   np.matmul(np.transpose(xTilde), y))
```

The normal equation has far higher accuracies than Batch GD with zero weight decay (Table 3). This is expected, since the normal equation is the analytic, optimal solution. The final mean square error (MSE) of the normal equation is also far lower than that of Batch GD. Lastly, the normal equation was computed several orders of magnitude quicker than Batch GD (0.4s versus 13.7s). The normal equation is the clear best choice in this case, as it outperforms Batch GD in nearly all metrics.

However, in a massive data set, Batch GD will perform much better in terms of runtime. This is because finding the normal equation requires taking the inverse of a matrix, which is extremely expensive computationally.

Table 3: Accuracy of normal equation vs. Batch GD

	Batch GD	Normal Equation
MSE	0.3901	0.01870
Training data accuracy	0.7917	0.9937
Validation data accuracy	0.76	0.96
Test data accuracy	0.7379	0.9448
Computation Time	13.752	0.438s

2 Logistic Regression

2.1 Loss Function and Gradient

In Logistic Regression, $\hat{y}$ is "squashed" with a sigmoid function:

$$\hat{y} = \sigma(W^T\mathbf{x} + b) \tag{3}$$

The cross entropy loss used in logistic regression, plus regularization, is then defined as:

$$\begin{aligned}\mathcal{L} &= \mathcal{L}_D + \mathcal{L}_W \\ &= \sum_{n=1}^{N} \frac{1}{N}[-y^{(n)}log(\hat{y})(\mathbf{x}^{(n)}) - (1-y^{(n)})log(1-\hat{y}(\mathbf{x}^{(n)}))] + \frac{\lambda}{2}\left\|W\right\|_2^2\end{aligned} \tag{4}$$

With a total of N samples and d features, we note that by defining $\mathbf{X}$ as the $N \times D$ data matrix, and $\vec{y}$ as a $N \times 1$ vector, we are able to vectorize the calculations of $\mathcal{L}_D$. The implementation is shown below:

```
def crossEntropyLoss(W, b, x, y, reg):
    y_linear_pred = np.matmul(x,W) + b
    y_pred = 1.0 / (1.0 + np.exp((-1)*y_linear_pred))
    Ld = np.sum((-1)*y*np.log(y_pred) - (1-y)*np.log(1-y_pred)) / x.shape
    [0]
    Lw = reg / 2 * (np.linalg.norm(W)**2)
    return Ld + Lw
```

Next, we attempt to calculate the gradient of the loss function, for one given sample (i.e. N = 1). $\mathcal{L}_W$ is still the same as before. To calculate $\mathcal{L}_D$, first, we compute the partial derivatives:

$$\frac{\partial \mathcal{L}_\mathcal{D}}{\partial \hat{y}} = \frac{-y}{\hat{y}} + \frac{1-y}{1-\hat{y}} \tag{5}$$

$$\frac{\partial \hat{y}}{\partial W} = \hat{y}^2(\frac{1}{\hat{y}} - 1)x = \hat{y}x - \hat{y}^2x \tag{6}$$

$$\frac{\partial \hat{y}}{\partial b} = \hat{y}^2(\frac{1}{\hat{y}} - 1) = \hat{y} - \hat{y}^2 \tag{7}$$

Using chain rule:

$$\frac{\partial \mathcal{L}_\mathcal{D}}{\partial W} = \frac{\partial \mathcal{L}_\mathcal{D}}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial W} \quad = x(\hat{y} - y) \tag{8}$$

$$\frac{\partial \mathcal{L}_\mathcal{D}}{\partial b} = \frac{\partial \mathcal{L}_\mathcal{D}}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial b} \quad = \hat{y} - y \tag{9}$$

We can now write out final expressions for the gradients:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{1}{N}(\mathbf{X}^T(\hat{y} - y)) + \lambda W \tag{10}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_{n=1}^{N}(\hat{y_n} - y_n) \tag{11}$$

Implementing this with Numpy:

```
def gradCE(W, b, x, y, reg):
    y_linear_pred = np.matmul(x,W) + b
    y_pred = 1.0 / (1.0 + np.exp((-1)*y_linear_pred))
    Lw = np.matmul(np.transpose(x), y_pred - y) / x.shape[0] + reg*W
    Lb = np.sum(y_pred - y) / x.shape[0]
    return Lw, Lb
```

2.2 Learning

```
def grad_descent(W, b, x, y, alpha, epochs, reg, error_tol, lossType):
    if (lossType=='CE'):
        for _ in range(epochs):
            dW, db = gradCE(W, b, x, y, reg)
            W = W - alpha*dW
            b = b - alpha*db
            if (np.linalg.norm(alpha*dW)<=error_tol):
                break
        return W, b
    return
```

Running this gradient descent implementation on our dataset, while keeping track of training, validation, and testing losses and accuracy over epochs, we observe that the logistic

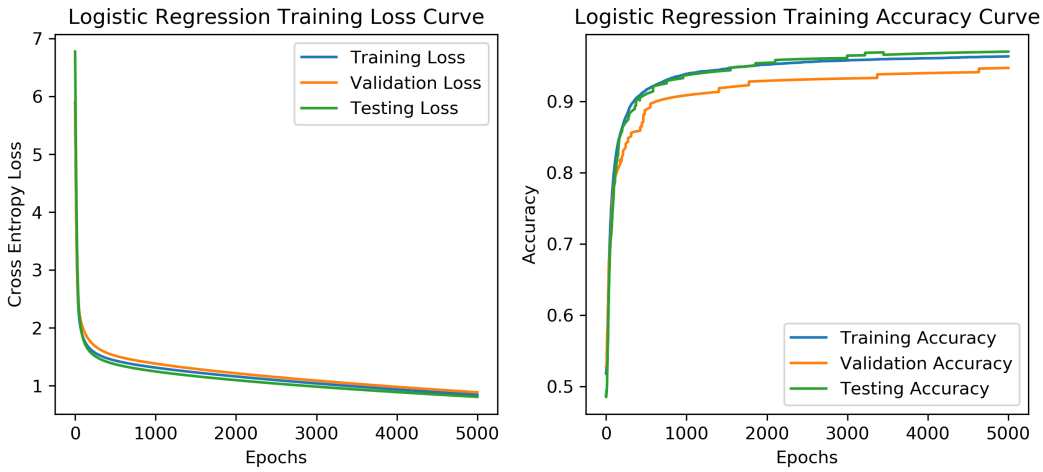

Figure 3: Training Loss and Accuracy Plot with Logistic Regression ($\alpha = 0.005, \lambda = 0.01$)

Table 4: Final Losses and Accuracies of Logistic Regression ($\alpha = 0.005, \lambda = 0.01$)

	Accuracy	Loss
Training Data	0.963	0.845
Validation Data	0.947	0.887
Test Data	0.970	0.809

regression model is able to fit well with the data and provide reasonable accuracy eventually. The training curves are shown here:

We also note the final loss and accuracy values, at the end of 5000 epochs of training.

2.3 Comparison to Linear Regression

Having implemented both models, we can now compare their performances. For a fair comparison, we test both models using $\alpha = 0.005$, $\lambda = 0.0$, over 5000 epochs.

Since the two models define losses differently, a even better comparison can be made with the accuracy plots.

Clearly from the training accuracy, the logistic model quickly converges (at around 1000 epochs) to an accuracy of more than 90%. On the other hand, the accuracy of linear model is still visibly increasing at the end of 5000 epochs of training, sitting only at around 80%. With the given set of parameters, the logistic model outperforms the linear model. One possible explanation is that logistic regression outputs numbers from 0 to 1, representing the probability the given set of inputs fits within one category. This matches closely with the task at hand: distinguishing the "C"s from the "J"s in the notMNIST dataset.

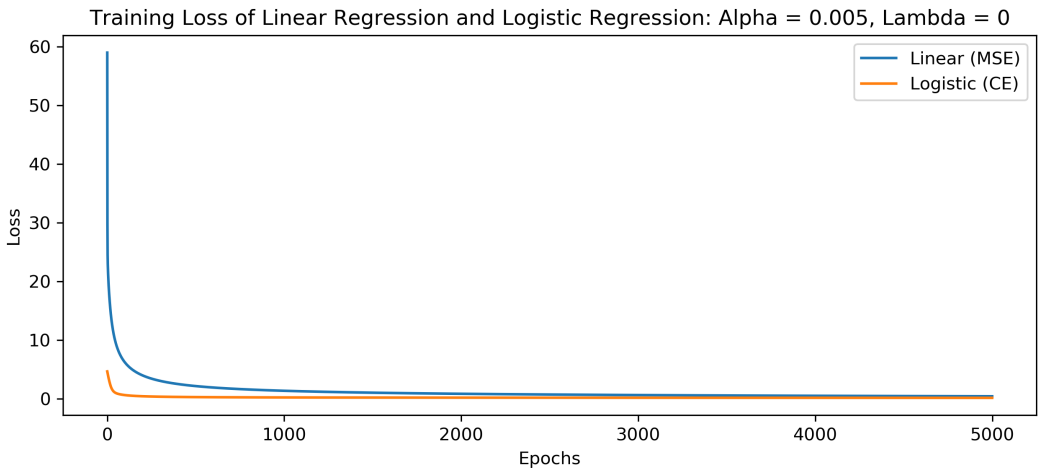

Figure 4: Comparison of Training Losses ($\alpha = 0.005, \lambda = 0.0$)

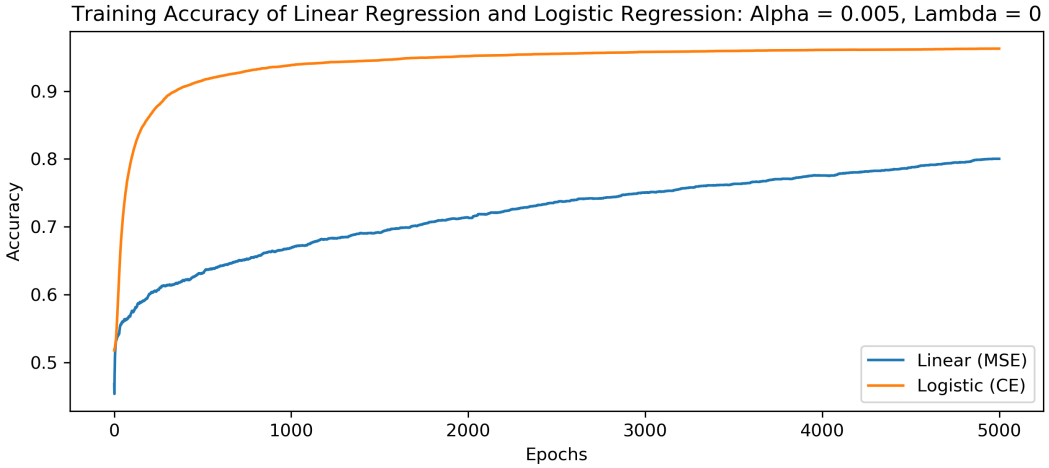

Figure 5: Comparison of Training Accuracy ($\alpha = 0.005, \lambda = 0.0$)

3 Batch Gradient Descent vs. SGD and Adam

3.1 Building the Computational Graph

For convenience, we combine building the graph and implementing SGD within the same function, the code of which is shown below:

```
def buildGraph(loss, b1, b2, eps):
    tf.set_random_seed(421)
    graph = tf.Graph()
    trainData_, validData_, testData_, trainTarget_, validTarget_, testTarget_ = loadData()
    trainData_, validData_, testData_ = map(reshape_data, (trainData_, validData_, testData_))

    with graph.as_default():

        # Initialize Weight and Bias Tensors
        W = tf.Variable(tf.random.truncated_normal(shape=[n_features, 1], mean=0.0, stddev=0.5))
        b = tf.Variable(tf.zeros(1))

        # Establish Placeholders for Data
        train_x = tf.placeholder(dtype='float32', shape=[n_batch_samples, n_features])
        train_y = tf.placeholder(dtype='float32', shape=[n_batch_samples, 1])
        validate_x = tf.placeholder(dtype='float32', shape=[n_validation_samples, n_features])
        validate_y = tf.placeholder(dtype='float32', shape=[n_validation_samples, 1])
        test_x = tf.placeholder(dtype='float32', shape=[n_test_samples, n_features])
        test_y = tf.placeholder(dtype='float32', shape=[n_test_samples, 1])

        if loss == "MSE":
            # Training Loss
            train_y_pred = tf.matmul(train_x, W)+b
            Ld = tf.losses.mean_squared_error(labels=train_y, predictions=train_y_pred)
            Lw = tf.nn.l2_loss(W)*reg #l2_loss output = sum(W ** 2) / 2
            L_train = Lw + Ld

            optimizer = tf.train.AdamOptimizer(learning_rate=alpha, beta1=b1, beta2=b2, epsilon=eps).minimize(L_train)

            # Validation Loss
            validate_y_pred = tf.matmul(validate_x, W)+b
```

```
            Ld = tf.losses.mean_squared_error(labels=validate_y, predictions=validate_y_pred)
            Lw = tf.nn.l2_loss(W)*reg #l2_loss output = sum(W ** 2) / 2
            L_validate = Lw + Ld

            # Testing Loss
            test_y_pred = tf.matmul(test_x, W)+b
            Ld = tf.losses.mean_squared_error(labels=test_y, predictions=test_y_pred)
            Lw = tf.nn.l2_loss(W)*reg #l2_loss output = sum(W ** 2) / 2
            L_test = Lw + Ld

        elif loss == "CE":
            # Training Loss
            train_y_pred = tf.sigmoid(tf.matmul(train_x, W)+b)
            Ld = tf.losses.sigmoid_cross_entropy(multi_class_labels=train_y, logits=train_y_pred)
            Lw = tf.nn.l2_loss(W)*reg
            L_train = Lw + Ld

            optimizer = tf.train.AdamOptimizer(learning_rate=alpha).minimize(L_train)

            # Validation Loss
            validate_y_pred = tf.sigmoid(tf.matmul(validate_x, W)+b)
            Ld = tf.losses.sigmoid_cross_entropy(multi_class_labels=validate_y, logits=validate_y_pred)
            Lw = tf.nn.l2_loss(W)*reg
            L_validate = Lw + Ld

            # Testing Loss
            test_y_pred = tf.sigmoid(tf.matmul(test_x, W)+b)
            Ld = tf.losses.sigmoid_cross_entropy(multi_class_labels=test_y, logits=test_y_pred)
            Lw = tf.nn.l2_loss(W)*reg
            L_test = Lw + Ld

    with tf.Session(graph=graph) as session:
        # Init
        tf.global_variables_initializer().run()
        tf.local_variables_initializer().run()

        n_batches = int(3500 / n_batch_samples)
        train_loss, train_accuracy = [], []
        validate_loss, validate_accuracy = [], []
        test_loss, test_accuracy = [], []

        # Train
        for epoch in range(n_epochs):
```

```
            trainData_, trainTarget_ = shuffle_data(trainData_,
    trainTarget_)
            for batch in range(n_batches):
                train_x_ = trainData_[n_batch_samples*batch:
    n_batch_samples*(batch+1), :]
                train_y_ = trainTarget_[n_batch_samples*batch:
    n_batch_samples*(batch+1), :]
                optimizer_, L_train_, L_validate_, L_test_, train_y_pred_,
     validate_y_pred_, test_y_pred_ = session.run(
                    [optimizer, L_train, L_validate, L_test, train_y_pred,
     validate_y_pred, test_y_pred],
                    {train_x: train_x_,
                     train_y: train_y_,
                     validate_x: validData_,
                     validate_y: validTarget_,
                     test_x: testData_,
                     test_y: testTarget_})
            train_loss.append(L_train_)
            train_accuracy.append(calculate_accuracy(train_y_,
    train_y_pred_))
            validate_loss.append(L_validate_)
            validate_accuracy.append(calculate_accuracy(validTarget_,
    validate_y_pred_))
            test_loss.append(L_test_)
            test_accuracy.append(calculate_accuracy(testTarget_,
    test_y_pred_))

        # Plot
        fig = plt.gcf()
        fig.set_size_inches(10, 4)

        plt.subplot(1,2,1)
        plt.plot(range(n_epochs),train_loss)
        plt.plot(range(n_epochs),validate_loss)
        plt.plot(range(n_epochs),test_loss)
        plt.legend(['Training Loss', 'Validation Loss', 'Testing Loss'])
        plt.ylabel('Loss')
        plt.xlabel('Epochs')
        plt.title('Training Loss Curve')

        plt.subplot(1,2,2)
        plt.plot(range(n_epochs),train_accuracy)
        plt.plot(range(n_epochs),validate_accuracy)
        plt.plot(range(n_epochs),test_accuracy)
        plt.legend(['Training Accuracy', 'Validation Accuracy', 'Test
    Accuracy'])
        plt.ylabel('Accuracy')
        plt.xlabel('Epochs')
        plt.title('Training Accuracy Curve')

```

```
116         plt.savefig(output_image_filename, dpi=300)
117
118     return W, b, train_y_pred, train_y, L_train, optimizer
```

Note that two helper functions have been defined:

```
def reshape_data(data):
    '''
    Reshapes data x to <n_samples, n_features>

    Input:
        Data <n_samples, features_x, features_y>

    Output:
        Data_aug <n_samples, features_x * features_y>,
        The data matrix, flattened.
    '''

    data_aug = np.reshape(data, (data.shape[0], data.shape[1]*data.shape
    [2]))
    return data_aug
```

```
def shuffle_data(x, y):
    randIndx = np.arange(y.shape[0])
    np.random.shuffle(randIndx)
    x_shuffled, y_shuffled = x[randIndx], y[randIndx]
    return x_shuffled, y_shuffled
```

3.2 Implementing Stochastic Gradient Descent

The implementation of SGD is integrated with the previous part.

3.3 Batch Size Investigation

Next, we can investigate the effect of batch size. Again, to keep the same basis for comparison, we set $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, $\lambda = 0.0$, over 700 epochs. The following results are obtained after running batch sizes of 100, 700, and 1750.

From both the loss and accuracy curves, it appears best performances can be achieved with smaller batch sizes. For example, using batch sizes of 100, convergence can be observed at around 500 epochs, at around an accuracy of above 0.9. However, for both instances using larger batch sizes (700 and 1750), convergence in accuracy is not clear from the accuracy plots. While the decay of loss slows down, accuracies are only around 0.8 and 0.65, respectively, at the end of training.

Theoretically, smaller batch size means gradients are calculated using fewer data points. Due to the variance in data points, this inevitably introduces some amount of noise (which

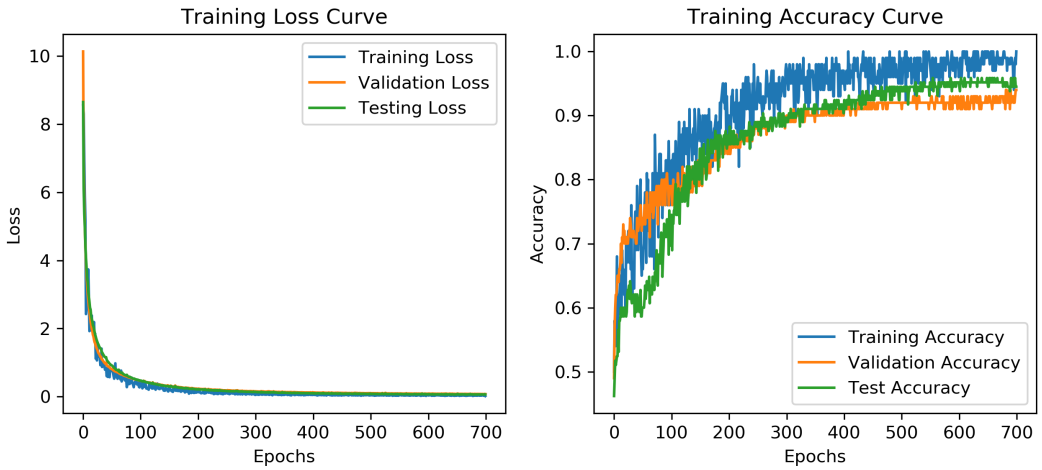

Figure 6: SGD Loss and Accuracy (Batches of 100)

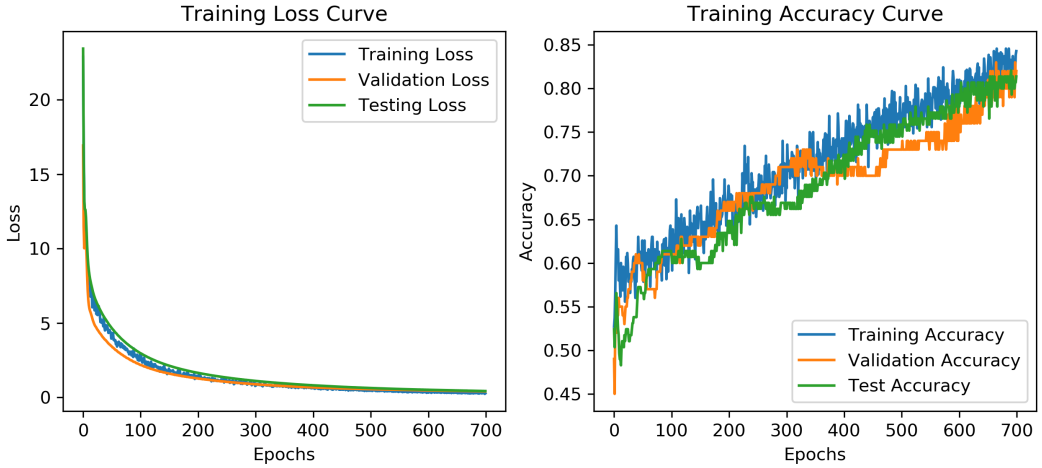

Figure 7: SGD Loss and Accuracy (Batches of 700)

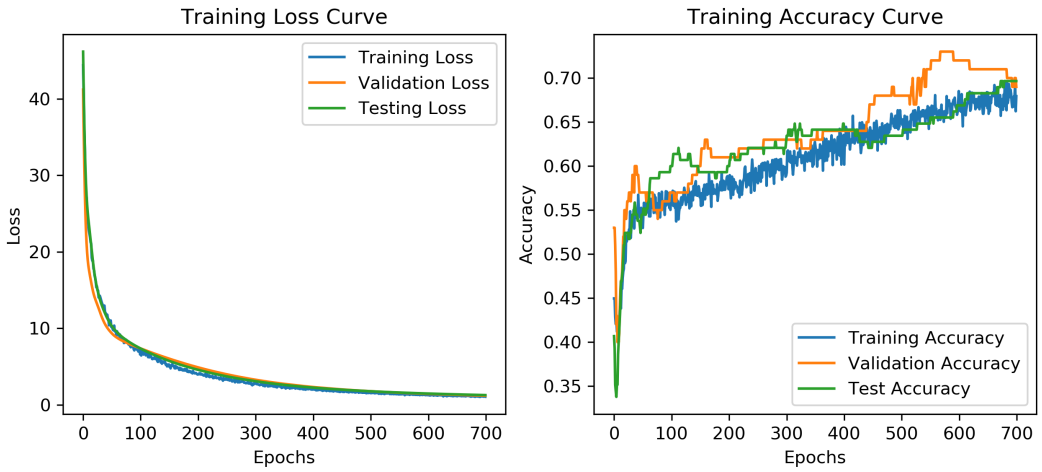

Figure 8: SGD Loss and Accuracy (Batches of 1750)

explains the rapid oscillations seen in the graph with batch sizes of 100), having adverse effects on overall training. However, in a well-labelled and generally known to be "easy" dataset such as notMNIST, it is reasonable to expect such variance to be small. In other words, our small batches of 100 samples are a sufficiently good representation of the entire training set. Using a smaller batch size however, allows us to update weights and biases many more times within 700 epochs. The addition number of times we are allowed to modify variables overweight the potential drawbacks that could have been caused by noise. Furthermore, the small amount of noise introduced might sometimes help with faster convergence and avoidance of local minimums. These two reasons might explain the better performance we observed with smaller batch sizes.

3.4 Hyperparameter Investigation

Table 5: Final MSE Accuracies with adjusted Hyperparameters

	$\beta_1 = 0.95$	$\beta_1 = 0.99$	$\beta_2 = 0.99$	$\beta_2 = 0.9999$	$\epsilon = 10^{-9}$	$\epsilon = 10^{-4}$
Training accuracy	0.92	0.862	0.948	0.864	0.904	0.814
Valid. accuracy	0.9	0.8	0.93	0.8	0.85	0.84
Test accuracy	0.8759	0.8552	0.9103	0.8414	0.9241	0.8483

a) The final learning accuracies for $\beta_1 = 0.95$ are higher than those for $\beta_1 = 0.99$ (Table 5). Additionally, in Figure 9, the accuracy curve for $\beta_1 = 0.99$ has significant oscillation during the early epochs. The documentation of the Adam hyperparameters (Figure 10) states that β_1 is the exponential decay rate for the first moment (mean) estimate. For each update step, β_1 is multiplied with the previous first moment value, and the current gradient is multiplied by $(1 - \beta_1)$. This means that higher values of β_1 place more weight on historic values rather than the current gradient. This adds a form of momentum to the model, which will help it to escape local minima. This would explain the oscillation when $\beta_1 = 0.99$, since the historic values in the early epochs are not as meaningful and may throw off the model.

b) The final learning accuracies for $\beta_2 = 0.99$ are higher than those for $\beta_2 = 0.9999$ (Table 5). The curves for $\beta_2 = 0.99$ and $\beta_2 = 0.9999$ in Figure 9 look similar, but the curve $\beta_2 = 0.9999$ has lower values than the curve for $\beta_2 = 0.99$. The documentation of the Adam hyperparameters (Figure 10) states that β_2 is the exponential decay rate for the second moment (uncentered variance) estimate. Similar to β_1, higher values of β_2 take more historic data into account, which may explain the lower accuracy.

c) The final learning accuracies for $\epsilon = 10^{-9}$ are higher than those for $\epsilon = 10^{-4}$ (Table 5). The plot for $\epsilon = 10^{-9}$ shows much faster learning near the beginning compared to $\epsilon = 10^{-4}$, where the learning rate stayed relatively constant (Figure 9). In the Adam documentation, ϵ is used to avoid division by zero in case the second moment estimate $(\hat{v}_t)$ is equal to zero. Since $\hat{v}_t$ is initialized to zero, the value of $\hat{v}_t$ in the early epochs may be very close to zero. In this case, small ϵ values contribute to a larger step size. This may explain why the model appeared to train quicker when ϵ took on the smaller value of 10^{-9}.

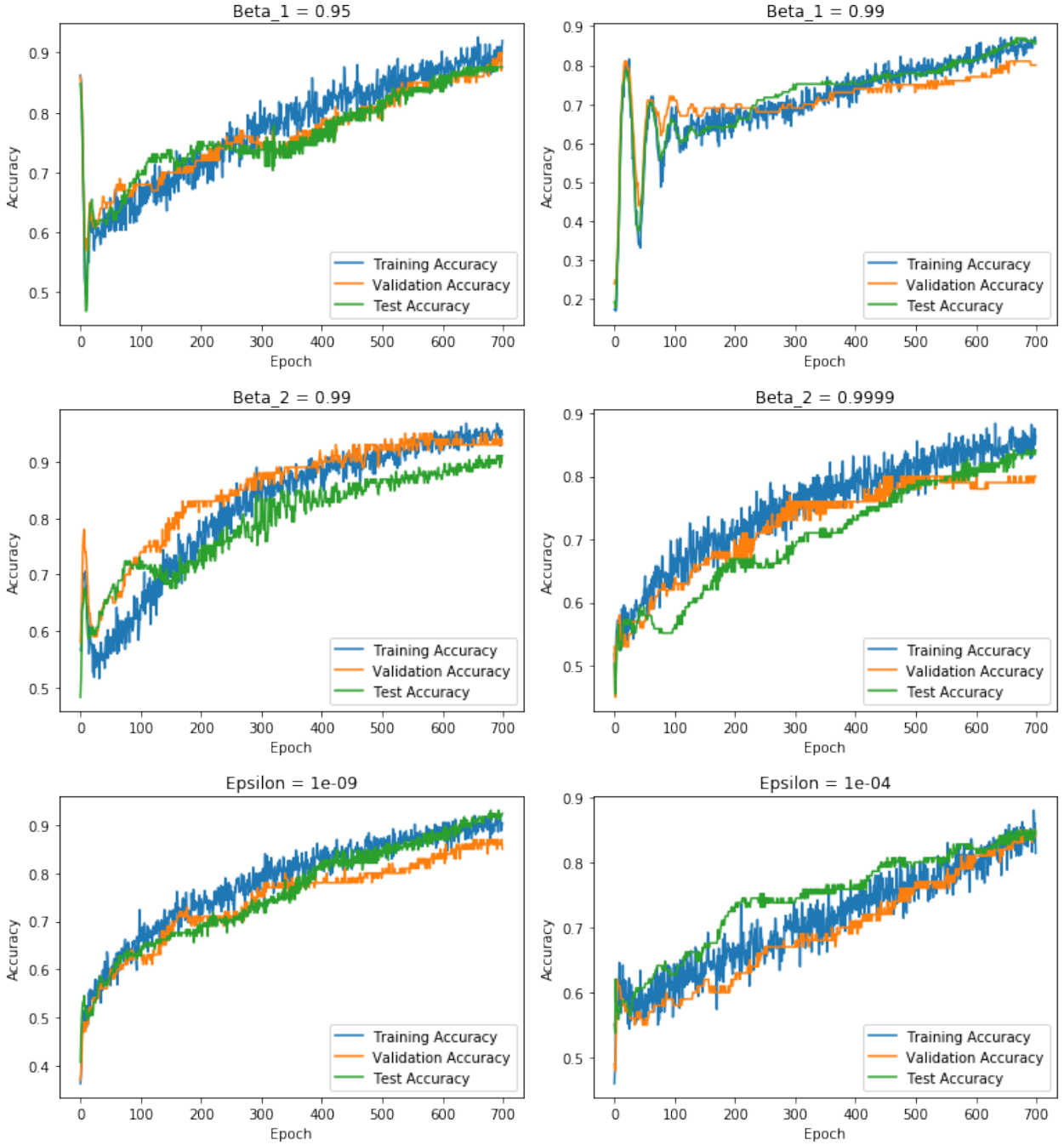

Figure 9: MSE Accuracy Curves with adjusted Hyperparameters

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Figure 10: Documentation of Adam hyperparameters from Adam paper [1].

Table 6: Final CE Accuracies with adjusted Hyperparameters

	$\beta_1 = 0.95$	$\beta_1 = 0.99$	$\beta_2 = 0.99$	$\beta_2 = 0.9999$	$\epsilon = 10^{-9}$	$\epsilon = 10^{-4}$
Training accuracy	0.99	0.99	0.994	0.992	0.988	0.988
Valid. accuracy	0.97	0.97	0.96	0.98	0.98	0.97
Test accuracy	0.9793	0.9793	0.9862	0.9655	0.9724	0.9793

3.5 Cross Entropy Loss Investigation

Similar to MSE, it appears that smaller batch sizes are more effective for CE. This may also be due to the well-labelled and simple nature of identifying "J"s and "C"s from notMNIST, where small batch sizes are representative of the entire data set. However, all the CE models trained much quicker and more accurately compared to the MSE models (Table 5 and Figures 11 and 12), with most nearing an accuracy of 1 in around 50 epochs. All the CE final accuracies were above 0.95 whereas none of the MSE accuracies reached 0.95. There are some disturbances in the $\epsilon = 10^{-4}$ curve, but this seems to be random jitter and did not affect the learning rate afterwards or the final accuracy. Changing the CE hyperparameters had very little to no effect on both the final accuracies and the accuracy curves (Table 5 and Figure 12). CE seems to be able to reach an optimized model well within 700 epochs, so it follows that small alterations to the hyperparameters would not greatly affect the learning rate or final accuracy.

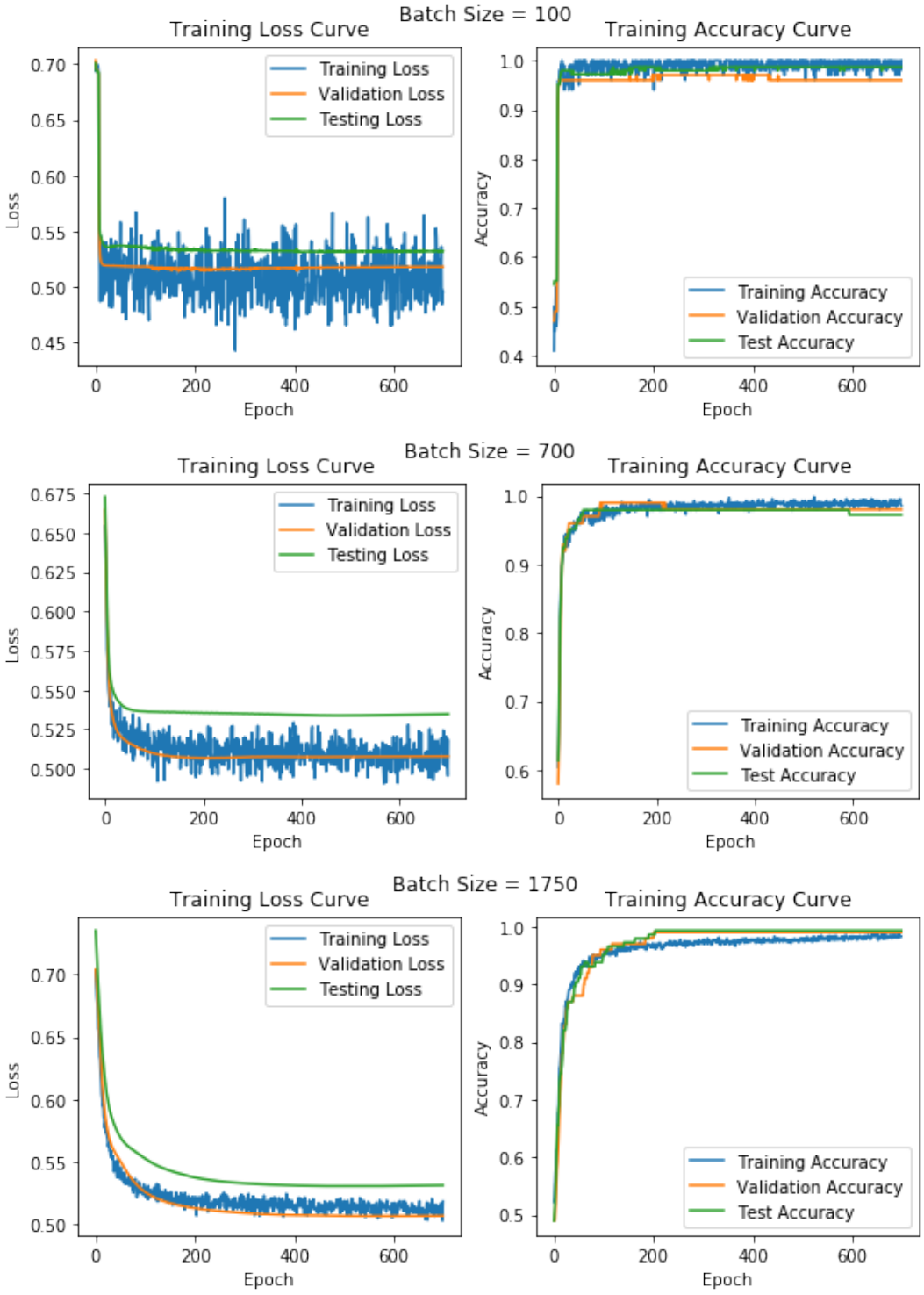

Figure 11: CE Accuracy Curves with Batch Sizes = 100, 700, and 1750

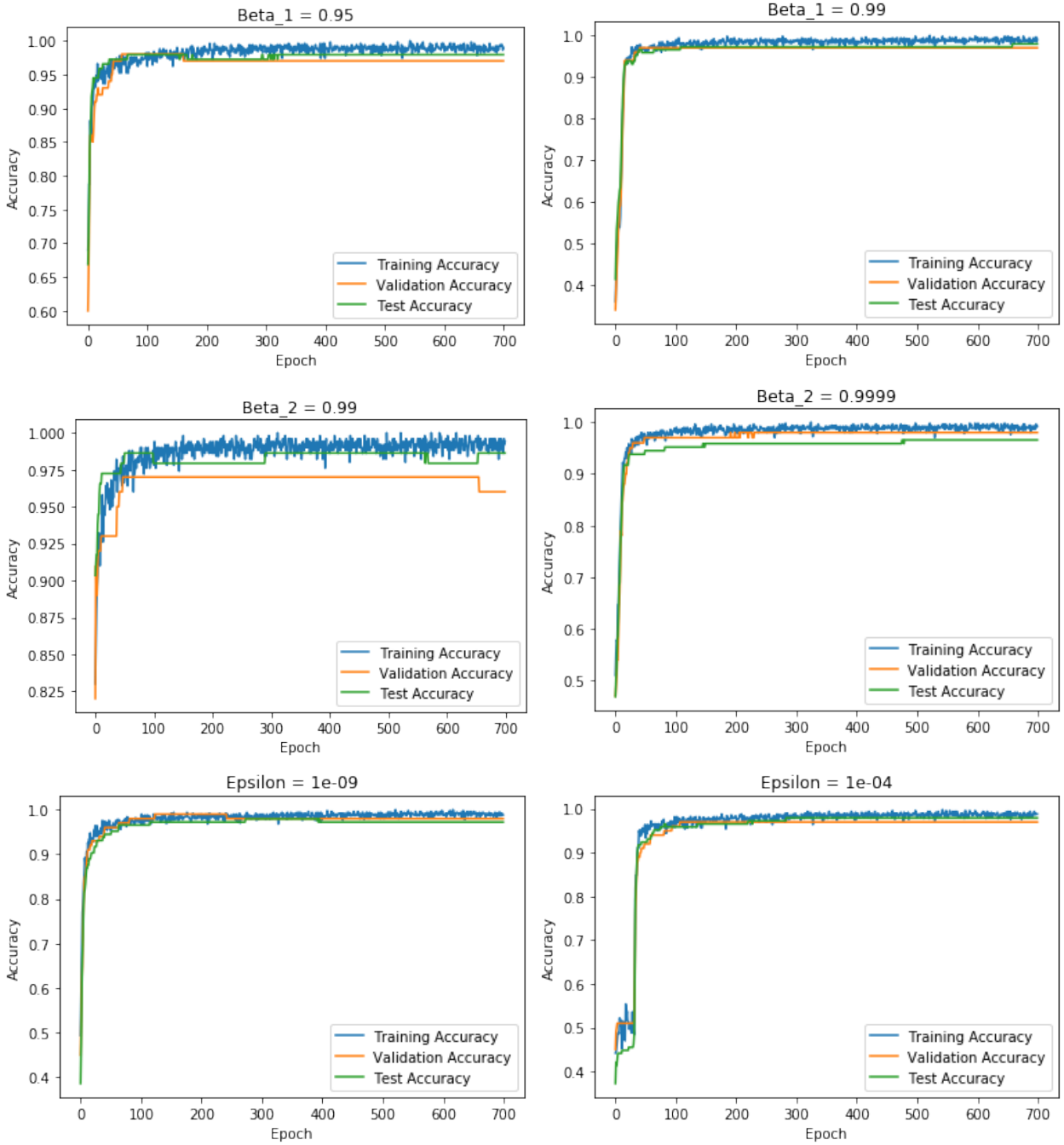

Figure 12: CE Accuracy Curves with adjusted Hyperparameters

3.6 Comparison against Batch GD

Overall, the SGD algorithm with Adam performs far better than the batch GD algorithm implemented earlier.

The best performing loss plot for MSE batch GD ($\alpha = 0.005$, $\lambda = 0.5$) neared zero around 500 epochs, with the other models taking over 2000 epochs to near zero (Figure 1 and 2). The MSE SGD models all neared zero loss by 700 epochs, with the batch size = 100 model only needing around 200 epochs (Figure 6, 7 and 8).

The loss plot for CE batch GD (Figure 3) took nearly 5000 epochs to converge near zero, whereas the loss plots for CE SGD dropped very quickly and neared zero within 300 epochs (Figure 11). The loss plot for CE SGD with a batch size of 100 may have even converged in nearly single digits epochs.

Many of the accuracy plots for MSE batch GD (1) all ended at relatively low accuracies (< 0.8) with a clearly positive slope. This means that they were not able to train sufficiently within 5000 epochs. Furthermore, there were many inconsistencies between the accuracies of the three data sets. The best performing MSE batch GD plot (again, $\alpha = 0.005$, $\lambda = 0.5$) took around 500 epochs to converge to an accuracy near one. The MSE SGD model achieved an accuracy of nearly one within 400 epochs, but the batch sizes of 700 and 1750 also exhibited lower final accuracies with a positive slope at the end. In general, the SGD graphs were a lot noisier, due to the reasons explained in section 3.3.

The accuracy plot for CE batch GD rose above 0.9 within 500 epochs and spent the rest of the 5000 epochs approaching it's final accuracy of 0.963 (3). In comparison, nearly all of the CE SGD models rose to nearly 100% accuracy within 200 epochs (Figures 11 and 12). SGD clearly outperformed batch GD here, both in terms of learning rate and final accuracy.

There are several possible reasons for SGD's superior performance over GD. The SGD algorithm with Adam is far more computationally efficient, as smaller sized batches result in less calculation time. The smaller batch sizes are also more effective when the data set is well-labelled and the task is simple, as in the case of identifying "J"s and "C"s from nonMNIST. Adam also includes momentum when determining weights, which helps the model to escape local minima.

References

[1] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization." 2014.