

# ECE521 Winter 2017: Assignment 2

FuYuan Tee, (999295837) \* Chee Loong Soon, (999295793) <sup>†</sup>

February 27th, 2017

## Contents

<b>1 Logistic Regression</b>	<b>3</b>
1.1 Binary cross-entropy loss . . . . .	3
1.1.1 Learning . . . . .	3
1.1.2 Beyond plain SGD . . . . .	5
1.1.3 Comparison with linear regression . . . . .	7
1.1.4 Maximum likelihood estimation . . . . .	11
1.2 Multi-class classification . . . . .	13
1.2.1 Minimum-loss system for equal misclassification penalty . . . . .	13
1.2.2 Minimum-loss system for unequal misclassification penalty . . . . .	16
1.2.3 Multi-class classification on <i>notMNIST</i> dataset . . . . .	19
<b>2 Neural Networks</b>	<b>21</b>
2.1 Geometry of neural networks . . . . .	21
2.1.1 Logistic regression weights on linearly separable dataset . . . . .	21
2.1.2 Logistic regression weights on linearly inseparable dataset . . . . .	22
2.1.3 Neural network weights on linearly inseparable dataset . . . . .	25
2.2 Feedforward fully connected neural networks . . . . .	29

---

\*Equal Contribution (50%), fuyuan.tee@mail.utoronto.ca

<sup>†</sup>Equal Contribution (50%), cheeloong.soon@mail.utoronto.ca

2.2.1	Layer-wise building block . . . . .	29
2.2.2	Learning . . . . .	30
2.2.3	Early stopping . . . . .	33
2.3	Effect of hyperparameters . . . . .	34
2.3.1	Number of hidden units . . . . .	34
2.3.2	Number of layers . . . . .	35
2.4	Regularization and visualization . . . . .	37
2.4.1	Dropout . . . . .	37
2.4.2	Visualization . . . . .	39
2.5	Exhaustive search for the best set of hyperparameters . . . . .	48
2.5.1	Random search . . . . .	48
2.5.2	Exchange ideas among the groups . . . . .	49
<b>3</b>	<b>Appendices</b>	<b>50</b>
3.1	Entire Code: Chee Loong Soon's version . . . . .	50
3.1.1	Part 1 . . . . .	50
3.2	Entire Code: FuYuan Tee's version . . . . .	100
3.2.1	Base Code for Q1 . . . . .	100
3.2.2	Q1.1.1: Learning . . . . .	109
3.2.3	Q1.1.2: Beyond Plain SGD . . . . .	111
3.2.4	Q1.1.3 Comparison with linear regression . . . . .	111
3.2.5	Q1.2.3: Multi-class classification . . . . .	115
3.2.6	Base Code for Q2 . . . . .	119
3.2.7	Q2.2 :Feedforward fully connected neural network . . . . .	136
3.2.8	Q2.3.1: Number of hidden units . . . . .	136
3.2.9	Q2.3.2: Number of layers . . . . .	136
3.2.10	Q2.4.1: Dropout . . . . .	137
3.2.11	Q2.4.2: Visualization . . . . .	137
3.2.12	Q2.5.1: Random search . . . . .	139
3.2.13	Q2.5.2: Exchange ideas among the group . . . . .	141

# 1 Logistic Regression

## 1.1 Binary cross-entropy loss

### 1.1.1 Learning

Using a two-class notMNIST dataset, a binary C-J character classifier was trained using logistic regression and cross-entropy loss. A TensorFlow Gradient descent optimizer was used to train the model. From using various learning rates, the best learning rate was selected to be  $\eta = 0.01$  due to its lowest validation loss and best test accuracy after 201 epochs (refer to Table 1 below).

Table 1: Performance of binary classifier with gradient descent optimizer with respect to  $\eta$

$\eta$	<i>Number of Iterations</i>	<i>Validation Loss</i>	<i>Best Test Accuracy, %</i>
0.001	1407	0.236	97.2
0.01	1407	0.138	100
0.1	1407	0.142	100
1	1407	0.793	98.6

The plots of cross-entropy loss and classification accuracy for the best learning rate can be found below:

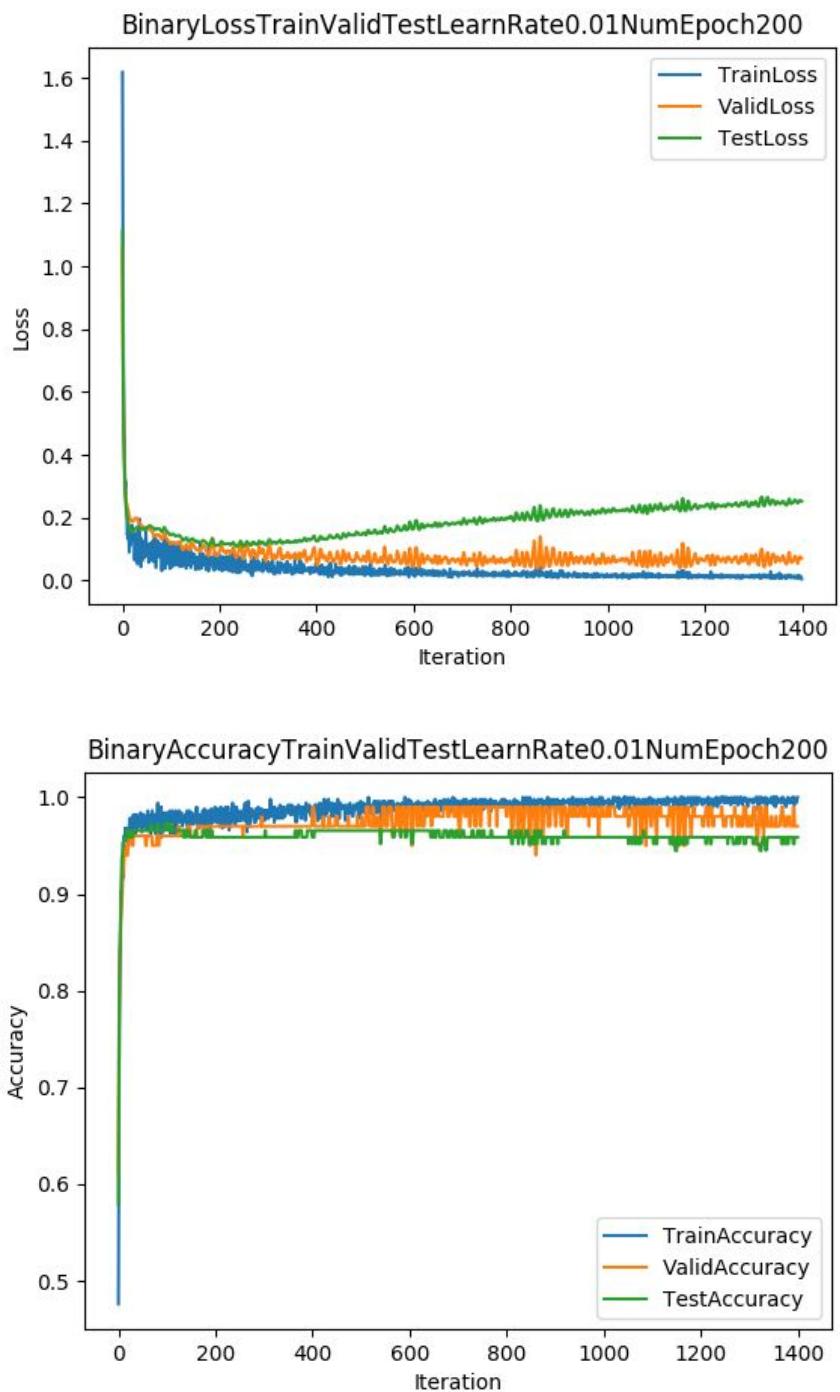


Figure 1: Training and test sets cross-entropy loss and classification accuracy for logistic classifier using stochastic gradient descent (SGD) and best learning rate,  $\eta = 0.01$ .

### 1.1.2 Beyond plain SGD

Similarly, a logistic binary classifier was trained on the same dataset using an Adam optimizer and early-stopping at the minimum validation loss in TensorFlow. The results are summarised in Table 2 below.

Table 2: Performance of binary classifier with Adam optimizer with respect to  $\eta$

$\eta$	Number of Iterations	Validation Loss	Best Test Accuracy, %
0.0001	1716	1.173	88.3
0.001	1471	0.327	96.6
0.01	253	0.298	97.9
0.1	15	1.397	97.9
1	43	2.115	97.9

The best learning rate was chosen to be  $\eta = 0.01$  as it has the highest test accuracy and lowest validation loss. Similar plots of cross-entropy loss and classification accuracy for the best learning rate can be found in Figure 2:

By comparing the performance of the binary classifier using both types of optimizers, it is noted that the test classification accuracies obtained are very similar. The only difference is that the Adam optimizer is able to allow the model to converge to the optimum weights at a much higher rate. Since this problem is convex, it helps the algorithm converge faster to the global minimum.

From the observation above, the Adam optimizer will prove to be useful in allowing for quicker convergence to more ideal solutions when learning from the *notMNIST* data. It also incorporates momentum to prevent the weights to get stuck in relatively poor local minimums for non-convex problems like part 2 of this assignment.

As the validation loss and test loss doesn't increase, it appears that the training has reached optimal weights and no longer trains the more epochs we go through, thus, leading to no increase in validation and test loss after a certain number of epochs.

Graphs of Best Learning Rate ( $\eta = 0.01$ ) using Adam optimizer

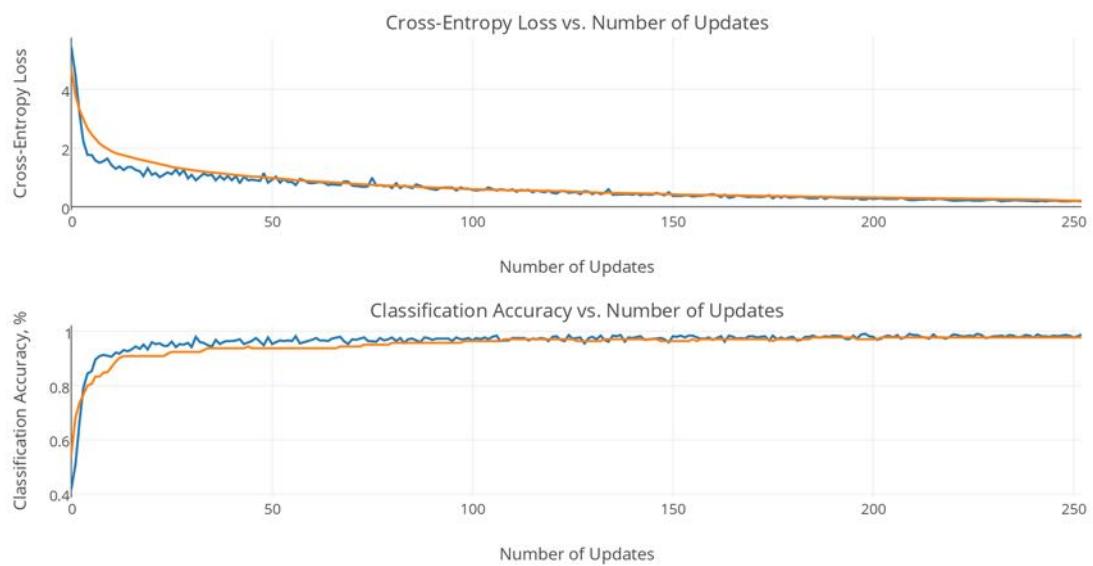


Figure 2: Training and test sets cross-entropy loss and classification accuracy for logistic classifier using Adam optimizer and best learning rate,  $\eta = 0.01$ .

### 1.1.3 Comparison with linear regression

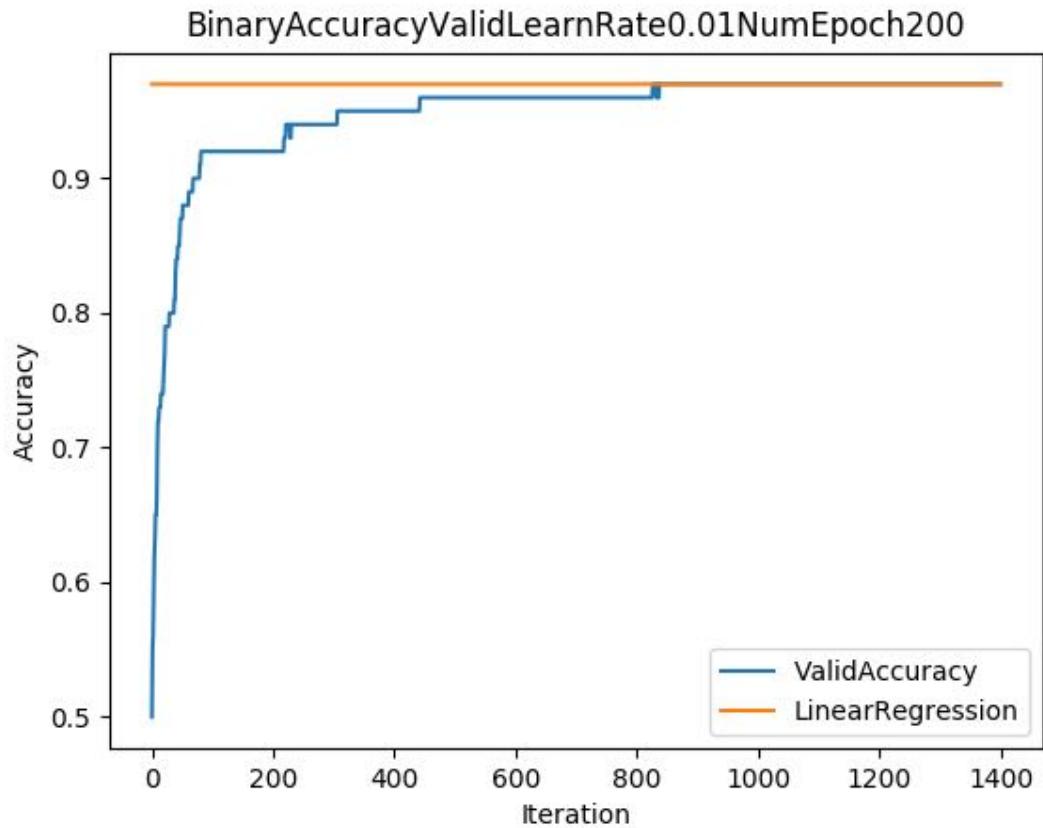


Figure 3: Comparison between Linear Regression Normal equation in yellow and cross-entropy loss in blue for validation classification accuracy

The cross entropy loss performs better than the optimal linear regression on the test set as it generalizes better than the optimal linear regression that was calculated using the training set as shown in figure 6.

Obviously, the optimal linear regression achieves a high accuracy on the training set as it was optimally trained on the training set as shown in figure 4.

The validation accuracies appears to converge to a similar value as shown in figure 3. This may be due to the simplicity of this problem where the optimal linear regression was able to generalize on the similar and small validation dataset compared to its training dataset.

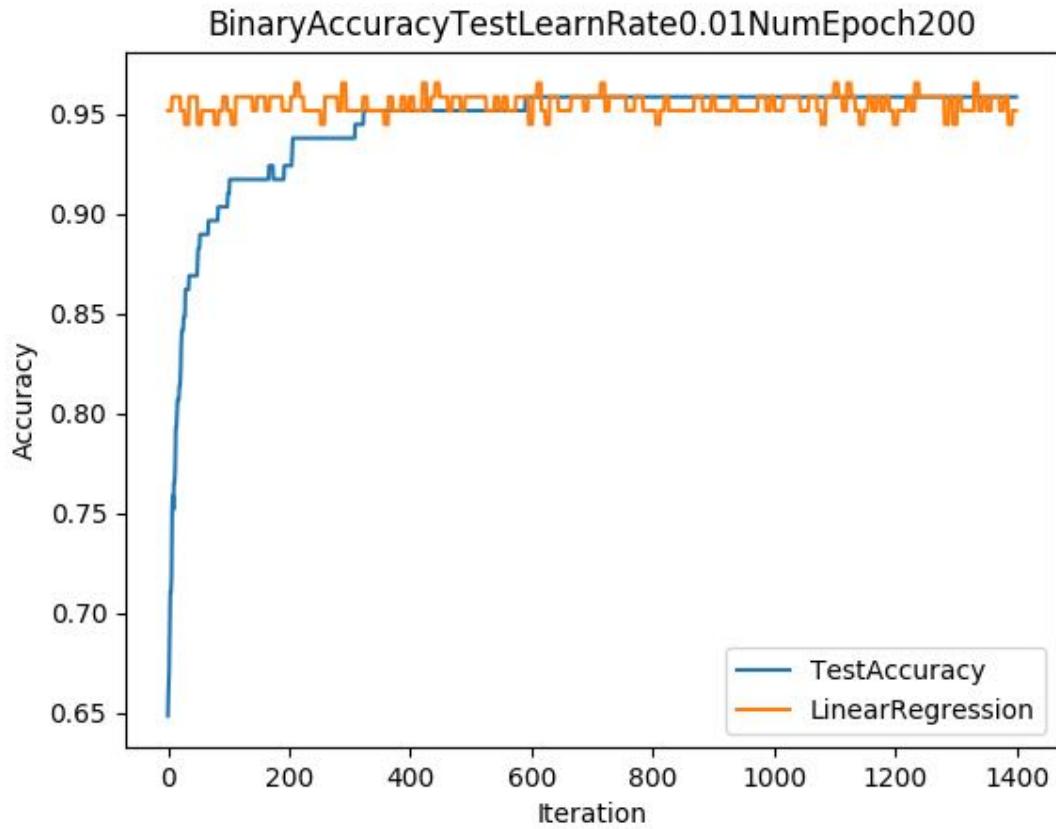


Figure 4: Comparison between Linear Regression Normal equation in yellow and cross-entropy loss in blue for test classification accuracy

To conclude, Linear Regression that is calculated using normal equation performs better on the training set since it is indeed the optimal weights for the training set. However, it overfits and performs worse on both the validation set and test set compared to the trained Logistic Regression model.

As seen from Figure 5, the cross-entropy loss heavily penalises misclassifications (i.e. cross-entropy loss approaches  $+\infty$  as  $\hat{y}$  approaches 1) as compared to a squared-error loss. This arises from the log term in the cross-entropy function, where  $\lim_{\hat{y} \rightarrow 1} \log(1 - \hat{y}) = +\infty$ . The fact that the cross-entropy loss penalises misclassifications significantly more is vital in minimising errors in classification applications.

Comparison of Cross Entropy and Squared Error Losses on Dummy Variable  $y = 0$

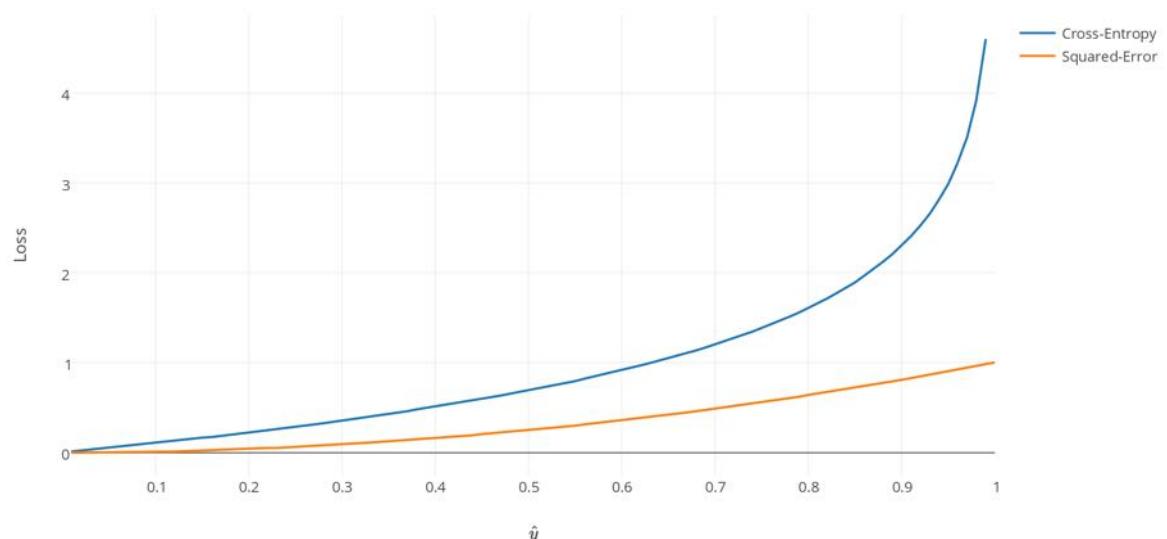


Figure 5: Comparison between squared-error and cross-entropy loss for  $\hat{y}$  against a dummy target  $y = 0$ .

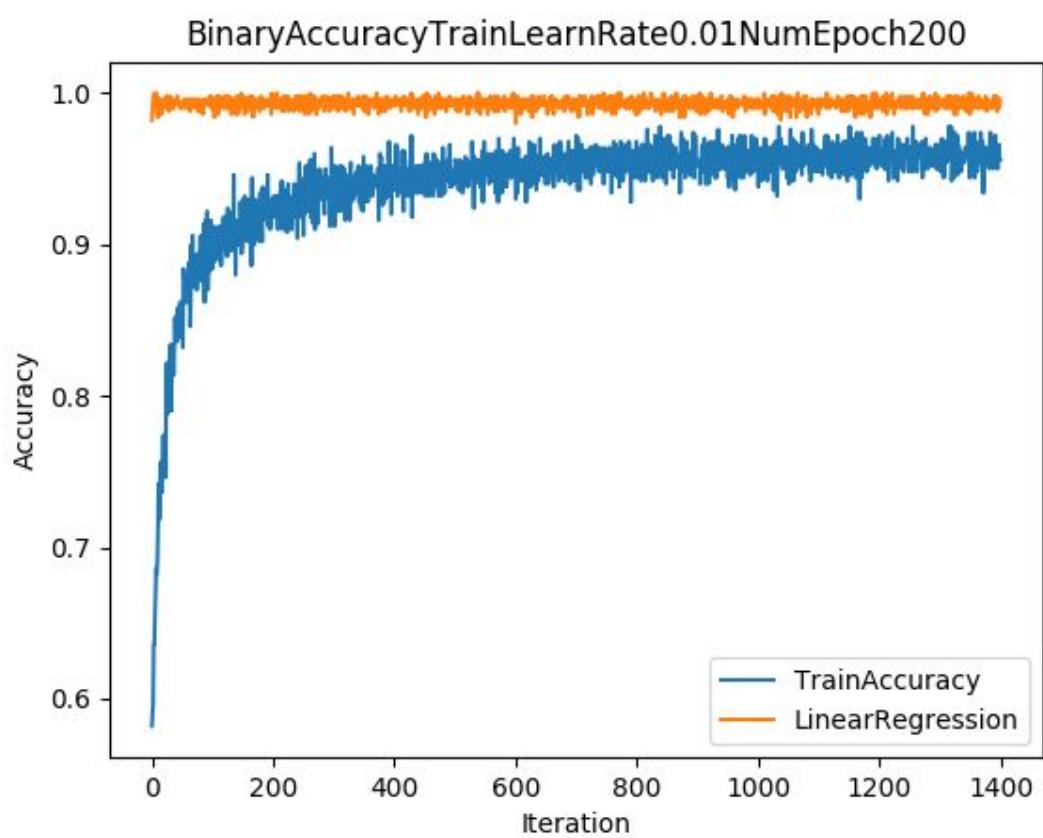


Figure 6: Comparison between Linear Regression Normal equation in yellow and cross-entropy loss in blue for training classification accuracy

### 1.1.4 Maximum likelihood estimation

Binary Classification means there are 2 outputs, 0 or w. In other words,  $y \in \{0, 1\}$ . Logistic function makes prediction of the outputs using equation 1. This equation makes the predicted value lie between [0, 1]. This equation assumes the bias is incorporated within the weights  $W$  where an additional dimension in the input  $x$  is filled with ones.

$$\hat{y}(\mathbf{x}) = \sigma(\mathbf{W}^T \mathbf{x}) = \frac{1}{1 + e^{-(\mathbf{W}^T \mathbf{x})}} \quad (1)$$

The combination of Cross Entropy Loss with Sigmoid Function is known as Logistic Regression shown in 2.

$$\sum_{m=1}^M \frac{1}{M} [-y^{(m)} \log(\hat{y}(\mathbf{x}^m)) - (1 - y^{(m)}) \log(1 - \hat{y}(\mathbf{x}^m))] \quad (2)$$

We can derive the cross entropy loss from Maximum Likelihood Estimation Principle by assuming a Bernoulli distribution for the likelihood of the training labels as shown in equation 3.

$$\Pr(y = 1|x, W) = 1 - \Pr(y = 0|x, W) = \hat{y}(x) \quad (3)$$

The Bernoulli Distribution for the likelihood can also be re-written more concisely as equation 4.

$$\Pr(y|x, W) = \hat{y}(x)^y (1 - \hat{y}(x))^{(1-y)} \quad (4)$$

Since maximizing the log-likelihood is same as maximizing the likelihood in equation 4, we take the log instead as it makes the computation easier as shown in equation 5.

$$\Pr(y|x, W) = y \log(\hat{y}(x)) + (1 - y) \log(1 - \hat{y}(x)) \quad (5)$$

Instead of maximizing the log-likelihood, we can instead minimize the negative log-likelihood by multiplying equation 5 with  $-1$  as shown in equation 6.

$$\Pr(y|x, W) = -y\log(\hat{y}(x)) - (1-y)\log(1-\hat{y}(x)) \quad (6)$$

Doing this for every  $M$  training case, where  $m$  represents a single training case, we get back to our original Logistic Regression equation 2.

Therefore, it is proven that maximizing the likelihood of training labels assuming a Bernoulli Distribution derives the Logistic Regression equation.

## 1.2 Multi-class classification

### 1.2.1 Minimum-loss system for equal misclassification penalty

If a classifier assigns each input,  $x$  to a class with largest a posteriori probability, in other words, assign  $x$  to class  $k$  if  $\Pr\{C_k|x\} \geq \Pr\{C_j|x\} \forall j \neq k$ . We can show that this represents a minimum-loss system if the penalty for misclassification is equal for each class.

The proof for minimum-loss for equal misclassification penalty is just a more specific case for a more general case for the proof for unequal misclassification penalty in section 1.2.2.

Equal misclassification penalty simply means a Loss Matrix where the diagonals are 0 or the trace of the loss matrix is 0. Also it means every non-diagonal elements would be an equal positive constant. For simplicity, we can assume that positive constant is 1.0.

This results in the Loss Matrix shown in matrix 7. The loss matrix is square matrix of size  $(K, K)$  where  $K$  represents the number of classes. The  $(j, k)$  entries of the loss matrix,  $L_{jk}$  represents the penalty for misclassification to a class  $j$  when the pattern in fact belongs to class  $k$ .

$$\mathbf{L} = \begin{bmatrix} 0 & 1 & \dots & \dots & 1 \\ 1 & 0 & 1 & \dots & 1 \\ \dots & \dots & \dots & \dots & \dots \\ 1 & \dots & \dots & 1 & 0 \end{bmatrix} \quad (7)$$

If you refer to the proof for section 1.2.2, you will understand that picking the class with largest a posteriori probability is the same as getting rid of the term with the largest posterior probability from the loss equation in 13.

This therefore represents the minimum loss for a specific region that the input belongs to. Repeat this for every input and you get a minimal-loss system for misclassification. This proofs this specific case where the penalty for misclassification is equal. However,

I will present a more specific proof below for that only works if the misclassification penalty is equal, resulting in the loss matrix shown in 7.

I will now explain why equation 19 represents picking the class with largest posterior probability for this specific case of equal misclassification penalty.

The a posterior probability of a class  $j$  is defined using Bayes Rule as shown in equation 8.

$$\Pr\{C_j|x\} = \frac{\Pr\{x|C_j\} \Pr\{C_j\}}{\sum_{k=1}^K \Pr\{x|C_k\} \Pr\{C_k\}} = \frac{\Pr\{x \in C_j\}}{\sum_{k=1}^K \Pr\{x|C_k\} \Pr\{C_k\}} \quad (8)$$

The denominator for the Bayes Rule in equation 8,  $\sum_{k=1}^K \Pr\{x|C_k\} \Pr\{C_k\}$  is simply a normalization constant.

Therefore, picking the class with largest posterior probability  $\Pr\{C_j|x\}$  is equivalent to picking the class with largest joint probability  $\Pr\{x \in C_j\}$ .

The error for misclassification of picking a class  $j$  is shown in equation 9. The first term is all the input in that region that does not belong to the assigned class  $j$  and the second term is all the input that does belong to class  $j$  but was not assigned to this class outside this region  $R_j$ .

$$\Pr\{mistake\} = \left( \sum_{k=1, k \neq j}^K \int_{R_j} \Pr\{x \in (R_j, C_k)\} \right) + \int_{R - R_j} \Pr\{x \in (R - R_j, C_j)\} \quad (9)$$

By definition of  $j$  being the largest posterior probability class, which is the largest joint probability class as shown in equation 10.

$$\begin{aligned} \Pr\{x \in (R_x, C_j)\} &\geq \Pr\{x \in (R_x, C_i)\} \\ &\forall i \neq j \end{aligned} \quad (10)$$

Thus, for an input that belongs to its region,  $R_x$  to be assigned to a class, picking the class with largest posterior probability is equivalent to picking the class with largest joint probability, which gets rid of that integral term from the sum of integrals in equation 9 in its region, therefore minimizes the loss and represents a minimum loss system. This further explained in the proof for section 1.2.2 below.

where equation 11 is a specific case of equation 20.

$$\arg \max_j \Pr\{C_k | \mathbf{x} \in R_j\} \quad (11)$$

### 1.2.2 Minimum-loss system for unequal misclassification penalty

Here we use a more general Loss Matrix shown in 12, where the entries can be any real number.

The loss matrix is square matrix of size  $(K, K)$  where  $K$  represents the number of classes. The  $(j, k)$  entries of the loss matrix,  $L_{jk}$  represents the penalty for misclassification to a class  $j$  when the pattern in fact belongs to class  $k$ .

$$\mathbf{L} = \begin{bmatrix} L_{11} & L_{12} & \dots & L_{1K} \\ L_{21} & L_{22} & \dots & L_{2K} \\ \dots & \dots & \dots & \dots \\ L_{K1} & \dots & \dots & L_{KK} \end{bmatrix} \quad (12)$$

We can define the entire domain of the entire input training data to be the entire region  $R$ . Given a specific region,  $R_{specific}$ , we need to classify any input in this specific region  $R_{specific}$  to a certain class. Lets just assume we assign every input in  $R_{specific}$  to a class  $\alpha$ . Then, we can re-define this specific region to be the assigned region  $R_\alpha$ .

To be clear, we define  $R_\alpha$  as the region where all inputs will be classified as class  $\alpha$ , where  $\alpha$  is a class in the  $K$  different classes. Obviously, we need to assign every domain of the region  $R$  to a class in the  $K$  classes. Otherwise, the loss penalty in 13 will be maximized and this is bad. However, we do not necessary need to assign to every single class that is available, although we must be using at least one of the  $K$  classes as explained.

We define  $C_k$  to be the class that a given input  $x$  actually belongs to. In this case, the input  $x$  belongs to the target class  $k$  and this cannot be changed by the algorithm as it is given as an input.

To be clear, we are only able to assign the regions  $R_{specific}$  to a class  $j$ . In other words  $R_{specific} = R_j$ . Only the inputs in this region  $R_{specific}$  will be classified as a result of this assignment. This input can be written as  $x \in (R_{specific}, C_k)$ . However, the input in this region  $R_{specific}$  already belongs to its actual class  $C_k$ .

Therefore, the penalty for the classification of a region  $R_{specific} = R_j$  will be  $L_{jk}$ . This results in  $x$  being assigned to class  $j$  but actually belongs to class  $k$ .  $j = k$  implies correct classification assignments whereas  $j \neq k$  implies misclassification. This is shown as  $x \in (R_j, C_k)$ .

The penalty for misclassification for entire region  $R$  for all  $K$  classes is defined in equation 13.

$$\mathbf{E}[Error] = \sum_{k=1}^K \sum_{j=1}^K L_{jk} \Pr\{x \in (R_j, C_k)\} \quad (13)$$

The total loss equation 13 sums over region for every assigned class  $R_j$ . We can simplify this to represent the loss equation for a single region assignment,  $R_{specific}$  as shown in 14.

$$\mathbf{E}[Error|R_{specific}] = \sum_{k=1}^K L_{specific,k} \Pr\{x \in (R_{specific}, C_k)\} \quad (14)$$

For a specific region,  $R_{specific}$ , the inputs in this region can belong to any of the  $K$  classes. We represent these inputs as a vector shown in 15.

$$\mathbf{X}_{R_{specific}} = \begin{bmatrix} x \in (R_{specific}, C_1) \\ x \in (R_{specific}, C_2) \\ \dots \\ x \in (R_{specific}, C_K) \end{bmatrix} \quad (15)$$

Therefore, selecting a region for  $R_{specific}$  is same as selecting a row  $j$  from the Loss Matrix in 12 to be dot product with the input vector 15.

The selected row  $j$  from the Loss Matrix is shown in 16.

$$\mathbf{L}_j = [L_{j1} \ L_{j2} \ \dots \ L_{jK}] \quad (16)$$

Thus, equation 14 can be represented as this dot product as shown in 17

$$\Pr\{Error|R_j\} = \mathbf{L}_j \cdot \mathbf{X}_{R_{specific}} = \mathbf{L}_j \cdot \mathbf{X}_{R_j} \quad (17)$$

Therefore, the condition to minimize the penalty for misclassification is to select the row  $j$  such that we minimize the dot product in 17. This is shown in equation 18

$$\arg \min_j \Pr\{Error|R_j\} = \arg \min_j \mathbf{L}_j \cdot \mathbf{X}_{R_{specific}} = \arg \min_j \mathbf{L}_j \cdot \mathbf{X}_{R_j} \quad (18)$$

This equation can be re-written as equation 19. I replaced the joint probabilities with posterior probabilities since it does not change the  $j$  that is picked as explained on the paragraph below equation 8.

$$\arg \min_j \Pr\{Error|R_j\} = \arg \min_j \sum_{k=1}^K \mathbf{L}_{jk} \Pr\{C_k|\mathbf{x} \in R_{specific}\} = \arg \min_j \sum_{k=1}^K \mathbf{L}_{jk} \Pr\{C_k|\mathbf{x} \in R_j\} \quad (19)$$

Therefore, the condition for minimum classification for an input in the region  $R_{specific}$  to a class  $C_j$  to minimize total loss is to pick the class  $j$  that results in the lowest  $\sum_{k=1}^K \mathbf{L}_{jk} \Pr\{C_k|\mathbf{x} \in R_{specific}\}$  for the region that the input belongs to.

For the special case where the diagonals of 12 is 0. i.e.  $L_{ii} = 0 \forall i$  and the other values positive (this means the loss matrix must penalize and there are no rewards (i.e. no negative values) for any classification). This means the dot product in 17 gets rid of one of the terms in the input vector 15 and it simply gets rid of that term in the sum in equation 14. We therefore pick the class  $j$  with the largest value for equation 20.

$$\arg \max_j \mathbf{L}_{jk} \Pr\{C_k|\mathbf{x} \in R_j\} \quad (20)$$

to get rid of that largest term,  $\mathbf{L}_{jk} \Pr\{C_k|\mathbf{x} \in R_j\}$  from the loss equation in 14. This will minimize the loss.

### 1.2.3 Multi-class classification on *notMNIST* dataset

For the full *notMNIST* dataset, a multi-class softmax classifier with Adam optimizer and early-stopping is implemented. The training results over various learning rates are summarised in Table 3 below.

Table 3: Performance of multi-class classifier with respect to  $\eta$

$\eta$	Number of Iterations	Validation Loss	Best Test Accuracy, %
0.0001	8371	1.060	84.1
0.001	1921	0.867	85.3
0.01	451	0.807	85.6
0.1	121	2.091	87.3
1	121	18.852	87.3

The best learning rate was chosen to be  $\eta = 0.01$  as it has the lowest validation error. The performance of that model can be seen from Figure 7 below.

The training and test classification performance is worse than the binary-class problem. This is because multi-class classification is a much more difficult problem due to the existence of more classes to predict and learn from with using the same logistic regression algorithm that may not be powerful enough to be able to learn the model. Thus, we will use Neural Networks in part 2 for this assignment to solve the multi-classification case.

Graphs of Multiclass Classification with Best Learning Rate ( $\eta = 0.01$ )

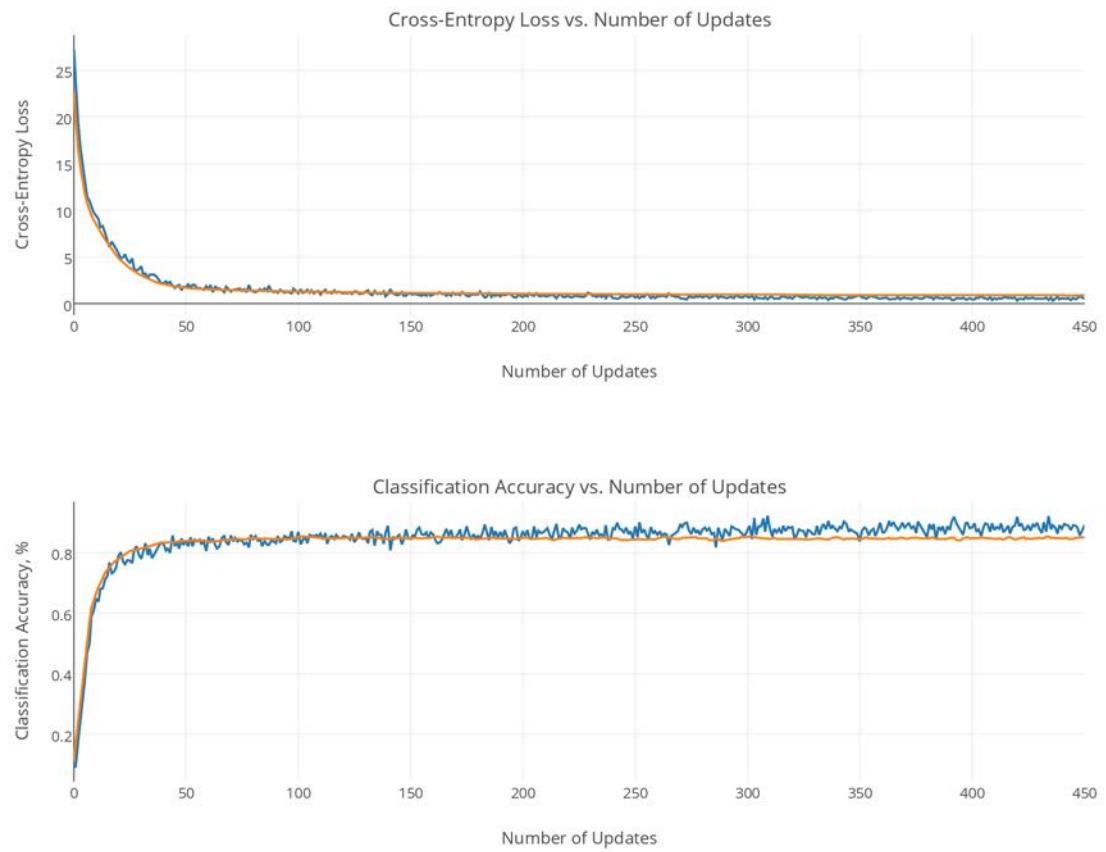


Figure 7: [Training](#) and [test](#) sets cross-entropy loss and classification accuracy for the softmax classifier with best learning rate,  $\eta = 0.01$ .

## 2 Neural Networks

### 2.1 Geometry of neural networks

#### 2.1.1 Logistic regression weights on linearly separable dataset

From the cross-entropy loss function:

$$\mathcal{L} = -\frac{1}{M} \sum_m^M [t^{(m)} \log y^{(m)} + (1 - t^{(m)}) \log (1 - y^{(m)})] \quad (21)$$

Let the bias,  $b$ , be lumped in as the zeroth weight,  $w_0$  in the weight vector,  $\mathbf{W}$ .

Taking the derivative with respect to the weights,  $\mathbf{W}$ :

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mathbf{W}} &= \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{W}} \\ &= -\frac{1}{M} \sum_m^M \left[ \frac{t^{(m)}}{y^{(m)}} - \frac{1 - t^{(m)}}{1 - y^{(m)}} \right] \cdot [y^{(m)} (1 - y^{(m)})] \cdot \mathbf{x}^{(m)'} \\ &= -\frac{1}{M} \sum_m^M [t^{(m)} (1 - y^{(m)}) - (1 - t^{(m)}) y^{(m)}] \cdot \mathbf{x}^{(m)'} \\ &= \frac{1}{M} \sum_m^M [y^{(m)} - t^{(m)}] \cdot \mathbf{x}^{(m)'} \\ &= \frac{1}{M} \sum_m^M \left[ \frac{1}{1 + e^{-(\mathbf{W}' \mathbf{x}^{(m)})}} - t^{(m)} \right] \cdot \mathbf{x}^{(m)'} \end{aligned} \quad (22)$$

Let the sets  $\mathbb{X}_0 = \{\mathbf{x}_0^{(1)}, \dots, \mathbf{x}_0^{(M_0)}\}$  and  $\mathbb{X}_1 = \{\mathbf{x}_1^{(1)}, \dots, \mathbf{x}_1^{(M_1)}\}$  be the sets of inputs with target values of  $t_0 = 0$  and  $t_1 = 1$  respectively, where  $M = M_0 + M_1$ . The partial derivative above can then be broken down into terms involving  $\mathbb{X}_0$  and  $\mathbb{X}_1$ , as follows:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{1}{M} \left\{ \sum_{m_0}^{M_0} \left[ \frac{1}{1 + e^{-(\mathbf{W}' \mathbf{x}_0^{(m)})}} - t_0^{(m)} \right] \mathbf{x}_0^{(m)'} + \sum_{m_1}^{M_1} \left[ \frac{1}{1 + e^{-(\mathbf{W}' \mathbf{x}_1^{(m)})}} - t_1^{(m)} \right] \mathbf{x}_1^{(m)'} \right\} \quad (23)$$

$$= \frac{1}{M} \left\{ \sum_{m_0}^{M_0} \left[ \frac{1}{1 + e^{-(\mathbf{W}' \mathbf{x}_0^{(m)})}} \right] \mathbf{x}_0^{(m)'} + \sum_{m_1}^{M_1} \left[ \frac{1}{1 + e^{-(\mathbf{W}' \mathbf{x}_1^{(m)})}} - 1 \right] \mathbf{x}_1^{(m)'} \right\}$$

$$= \frac{1}{M} \left\{ \sum_{m_0}^{M_0} \left[ \frac{1}{1 + e^{-(\mathbf{W}' \mathbf{x}_0^{(m)})}} \right] \mathbf{x}_0^{(m)'} + \sum_{m_1}^{M_1} \left[ \frac{1}{1 + e^{\mathbf{W}' \mathbf{x}_1^{(m)}}} \right] \mathbf{x}_1^{(m)'} \right\} \quad (24)$$

A dataset is linearly separable when there exists  $w_1, \dots, w_n, k \in \mathbb{R}$  such that  $\sum_i^n w_i x_i < k \quad \forall x_i \in \mathbb{X}_0$  and  $\sum_i^n w_i x_i > k \quad \forall x_i \in \mathbb{X}_1$ , using the sets definition for  $\mathbb{X}_0$  and  $\mathbb{X}_1$  above. The dataset is linearly separable. By the property of linearly separable, it means that it is possible to select values for the optimal weight vector  $\mathbf{W}^*$ , such that:

$$\begin{cases} \mathbf{W}^{*\prime} \mathbf{x}_0^{(m)} \ll 0 \\ \mathbf{W}^{*\prime} \mathbf{x}_1^{(m)} \gg 0 \end{cases} \quad \text{or} \quad \begin{cases} \mathbf{W}^{*\prime} \mathbf{x}_0^{(m)} = -\infty \\ \mathbf{W}^{*\prime} \mathbf{x}_1^{(m)} = +\infty \end{cases} \quad \forall m = 1, \dots, M \quad (25)$$

This in turn forces the coefficients of  $\mathbf{x}_0^{(m)'} \mathbf{x}_0^{(m)}$  and  $\mathbf{x}_1^{(m)'} \mathbf{x}_1^{(m)}$  in Equation 24 to be 0, leading to gradients for each individual training example to go to 0, leading to an optimal solution. Since the dataset are indicated to be linearly separable and that the weights are optimal, it means that the gradients must be 0 for any training data, which means the coefficients must go to 0, which means that the weights have magnitudes at infinity.

Since the inputs  $\mathbf{x}^{(m)}$  is finite, for Equation 25 to occur:

$$|w_i| = \infty \quad \forall i = 1, \dots, n \quad (26)$$

and by extension,  $\|\mathbf{W}\|_2 = \infty$ .

### 2.1.2 Logistic regression weights on linearly inseparable dataset

Note that the mathematical notation below is different from section 2.1.1. Here,  $y$  refers to the target output and  $\hat{y}$  refers to the predicted output from the sigmoid.

Rewriting cross entropy loss in 21 using this new notation, we get equation 27

$$L = -\frac{1}{M} \sum_m^M [y^{(m)} \log \hat{y}^{(m)} + (1 - y^{(m)}) \log (1 - \hat{y}^{(m)})] \quad (27)$$

Rewriting the gradient of cross entropy loss with respect to weights results in equation 28 where  $M$  is the number of training data.

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{1}{M} \sum_m^M \left[ \frac{1}{1 + e^{-(\mathbf{W}' \mathbf{x}^{(m)})}} - y^{(m)} \right] \cdot \mathbf{x}^{(m)} = \frac{1}{M} \sum_m^M [\hat{y}^m - y^{(m)}] \cdot \mathbf{x}^{(m)} \quad (28)$$

For single sigmoid neuron on a linearly inseparable binary classification problem, we need to show that the length of optimal weights vector learn on this dataset will always be bounded.

Lets clearly define how each term above is interpreted mathematically.

For a linearly inseparable case, Equation 25 cannot occur by definition. As a result, the coefficients of  $x_0^{(m)'} \text{ and } x_1^{(m)'}$  will never be zero themselves. This fact extends to the individual gradient of each training example being non-zero. As a result, the only way for an optimal solution to be reached is for the individual gradients of various training example canceling each other out to obtain an overall gradient value of 0. Similarly, the optimal solution is reached without having  $\mathbf{W}^* \mathbf{x}^{(m)} = \pm\infty$ . As a result,  $\|\mathbf{W}\|_2 < \infty$ .

Another way to proof this is by substituting it into the Cross Entropy Loss equation in 27 assuming the weights were bounded and linearly inseparable as shown below.

Binary Classification problem means the output can only be 2 values. For the case of a single sigmoid where it can only output values between 0 to 1 (i.e.  $\hat{y} \in [0, 1]$ ), it mathematically makes sense to define the 2 values for the binary classification to be  $\{0, 1\}$ .

As the weights are unbounded, the predictions of a sigmoid are always either 0 or 1 (i.e.  $\hat{y} \in \{0, 1\}$ ) The target labels are also 0 or 1 (i.e.  $y \in \{0, 1\}$ )

Therefore, the four cases or prediction and output are:  $(y, \hat{y}) = \{(0, 0), (1, 1), (0, 1), (1, 0)\}$

Substituting these four cases into the equation 27 and ignoring the sum over M training cases, we get equations shown in 29 and 30 .

$$\begin{aligned} L &= -[0 \log 0 + (1 - 0) \log(1 - 0)] = -[0 + 1 \log 1] = 0 \\ L &= -[1 \log 1 + (1 - 1) \log(1 - 1)] = -[1 \log 1 + 0 \log 0] = 0 \end{aligned} \tag{29}$$

$$\begin{aligned} L &= -[0 \log 1 + (1 - 0) \log(1 - 1)] = -[0 + 1 \log 0] = -[1 \log 0] = \text{undefined} \\ L &= -[1 \log 0 + (1 - 1) \log(1 - 0)] = -[1 \log 0 + 0 \log 1] = -[1 \log 0] = \text{undefined} \end{aligned} \tag{30}$$

However, the equations in 30 is undefined and cannot possible occur. That only leaves us with the possibility of equations 29 occurring if the weights are unbounded. However, this only occurs when the input and outputs are correctly classified and hence are linearly separable leading to a contradiction that the dataset is linearly inseparable.

This proves that the optimal weights on a linearly inseparable dataset must be unbounded.

### 2.1.3 Neural network weights on linearly inseparable dataset

A Single Hidden Layer Neural Network can classify a linearly inseparable binary classification dataset since it is a sum of non-linear activations by the universality theorem.

Unbounded case, Before being fed into the single hidden layer neural network, each single sigmoid neuron will have bounded optimal weights due to the linearly inseparable dataset as proven in section 2.1.2. However, these inputs can then be transformed by the first hidden layer to be into linearly separable transformed inputs for the final output activation. The final output activation will be able to classify these linearly separable transformed inputs and will have unbounded optimal weights as proven in section 2.1.1. As the neural network  $\text{vec}\{W\}$  is a concatenation of both the weights from the first hidden layer and the final activation output, it will consist of both bounded and unbounded weights and the  $\text{vec}\{W\} = \inf$  (i.e. unbounded) for this globally optimal solution where it classifies each data correctly.

Bounded case, we can of course show that there exist some locally optimal solution where the input remains linearly inseparable and so the final layer will remain bounded. Hence, the  $\text{vec}\{W\} < \inf$  (i.e. bounded).

Consider a simple example of the famous XOR problem on 4 data points that is linearly inseparable. The data points are 2-Dimensional and have a 1-Dimensional binary output. Let  $\mathbf{x}^i$  be the  $i^{th}$  input where  $i \in \{1, 2, 3, 4\}$ , where  $\mathbf{x}^i = [x_1^i, x_2^i]$  since the input is 2-Dimensional. Define the 4 exact inputs as shown in Table 4.

Table 4: Input and Target Output Data

$X_2$	$X_1$	$y$
0	0	0
0	1	1
1	0	1
1	1	0

Using 2 hidden units in the first layer. Both with 2-Dimensional weights and 1-Dimensional bias.

first hidden unit has weights and bias  $\mathbf{W}^1 = [W_1^1, W_2^1]$  and  $b^1$

second hidden unit has weights and bias  $\mathbf{W}^2 = [W_1^2, W_2^2]$  and  $b^2$

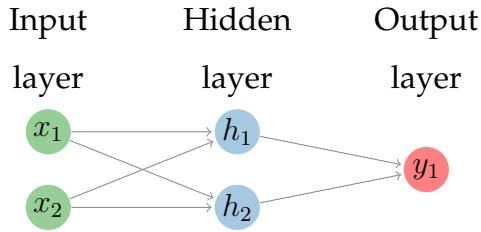
Final output activation has weights 2-Dimension weights and 1-Dimensional bias  $\mathbf{W}^3 = [W_1^3, W_2^3]$  and  $b^3$

Output activation of the hidden units is defined in 1.

The output activation of the output layer is said to be a step function where it outputs 1 if the input is  $> 0$  and 0 if the input is  $\leq 0$

Optimal weights means that the weights are at a local minimum and its gradient is 0. We can also say that an input data that is perfectly classified has no loss and therefore zero gradient.

The simple neural network with 2 hidden units in a single hidden layer with a 2 dimensional input is shown below.



These are all related as shown below in equation 31

$$\begin{aligned}
 z_1 &= \mathbf{W}^1 \cdot \mathbf{X}^T + b^1 \\
 h_1 &= \sigma(z_1) \\
 z_2 &= \mathbf{W}^2 \cdot \mathbf{X}^T + b^2 \\
 h_2 &= \sigma(z_2) \\
 \mathbf{h} &= [h_1, h_2] \\
 \hat{y} &= \text{step}(\mathbf{W}^3 \cdot \mathbf{h}^T + b^3)
 \end{aligned} \tag{31}$$

The final prediction,  $\hat{y}$  in equation 31 is compared against the actual target  $y$  using the cross entropy loss function.

The first configuration in the equations 32 presented below results in all the points

correctly classified and the weights are all bounded. The weights and biases are all  $< \inf$  but it correctly classifies the 4 points. Therefore, it is an optimal solution that is bounded.

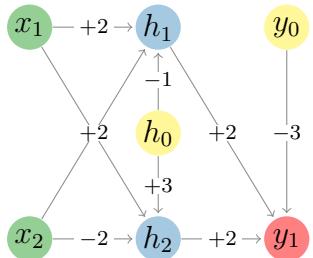
$$\begin{aligned}
\mathbf{W}^1 &= [W_1^1, W_2^1] = [2, 2] \\
b^1 &= -1 \\
\mathbf{W}^2 &= [W_1^2, W_2^2] = [-2, -2] \\
b^2 &= 3 \\
\mathbf{W}^3 &= [W_1^3, W_2^3] = [2, 2] \\
b^3 &= -3
\end{aligned} \tag{32}$$

The output for the equations in 31 using configuration 32 is shown in Table 5.

Table 5: Bounded Output Table

$X_2$	$X_1$	$y$	$z_1$	$z_2$	$h_1$	$h_2$	$\hat{y}$
0	0	0	-1	3	0	1	0
0	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1
1	1	0	3	-1	1	0	0

The illustration is shown below shows the Bounded configuration on the neural network. note that the link from  $x_1$  to  $h_2$  is covered and is supposed to be -2. In yellow are the bias links.



The second configuration in the equations 33 presented below results in all the points correctly classified and there exist unbounded weights. The weights  $\mathbf{W}^3 = \inf$  and it correctly classifies the 4 points. Therefore, it is an optimal solution that is unbounded.

$$\mathbf{W}^1 = [W_1^1, W_2^1] = [2, -1]$$

$$b^1 = -1$$

$$\mathbf{W}^2 = [W_1^2, W_2^2] = [-1, 2]$$

(33)

$$b^2 = -1$$

$$\mathbf{W}^3 = [W_1^3, W_2^3] = [\inf, \inf]$$

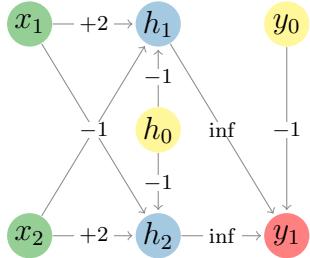
$$b^3 = -1$$

The output for the equations in 31 using configuration 33 is shown in Table 6 .

Table 6: UnBounded Output Table

$X_2$	$X_1$	$y$	$z_1$	$z_2$	$h_1$	$h_2$	$\hat{y}$
0	0	0	-1	-1	0	0	0
0	1	1	1	-2	1	0	1
1	0	1	-2	1	0	1	1
1	1	0	0	0	0	0	0

The illustration is shown below shows the UnBounded configuration on the neural network. In yellow are the bias links.



From both configurations above, this example shows that under cross entropy loss without weight decay, there are some locally optimal solutions with bounded weight vector  $\|vec\{W^*\}\|_2 < \inf$  and some other locally optimal solutions with unbounded weight vector  $\|vec\{W^*\}\|_2 = \inf$

## 2.2 Feedforward fully connected neural networks

### 2.2.1 Layer-wise building block

```
1  def layerWiseBuildingBlock(self, inputTensor,
2      ↵  numberOfHiddenUnits):
3          """
4              Layer-wise Building Block
5
6              Input:
7                  inputTensor: Hidden activations from previous
8                      ↵  layer
9                          numberOfHiddenUnitsInCurrentLayer
10
11             Output: z = Weighted sum of the inputs
12
13             Initialize Weight Matrix and Biases in this function
14
15             A list of hidden activations
16
17             No Loops
18
19             """
20
21         numInput = inputTensor.get_shape().as_list()[1]
22         numOutput = numberOfHiddenUnits
23
24         # Xavier Initialization
25
26         #variance = tf.div(tf.constant(3.0), tf.add(numInput,
27             ↵  tf.constant(numOutput)))
28
29         variance = 3.0 / (numInput + numOutput)
30
31         weight =
32
33             ↵  tf.Variable(tf.truncated_normal(shape=[numInput,
34                 ↵  numOutput], stddev = math.sqrt(variance)))
35
36         bias = tf.Variable(tf.zeros([numOutput]))
37
38         weightedSum = tf.matmul(tf.cast(inputTensor,
39             ↵  "float32"), weight) + bias
40
41         tf.add_to_collection("Weights", weight)
```

## 2.2.2 Learning

A neural network with hyperparameters listed in Table 7. ReLU functions were used for hidden layer activations while softmax was used for output layer activations. The model was trained using various learning rates, as summarised in Table 8.

Table 7: Neural network hyperparameters

<i>Hyperparameter</i>	<i>Value</i>
Batch size, $B$	500
L2 regularizer, $\lambda$	$3 \times 10^{-4}$
Hidden Layers	1
Hidden Units	(1000)
Dropout Rate	0%

Table 8: Performance of neural network with respect to  $\eta$

<i>Learning Rate, <math>\eta</math></i>	<i>Epochs</i>	<i>Validation Loss</i>	<i>Best Test Accuracy, %</i>
0.0001	21	0.354	91.5
0.001	5	0.330	91.9
0.01	7	0.588	91.4

The best learning rate was found to be  $\eta = 0.001$ , which produces the lowest validation loss and highest test accuracy.

The performance of this model can be seen in Figure 9.

The best performance accuracy is shown below in Figure 8. After 20 epochs, the training accuracy approaches 99.8 percent, validation approaches 93.8 percent and test accuracy approaches 91.2 percent accuracy and again summarize in Table 9

20 epochs was from Soon implementation. 5 epoch was from Tee implementation. Soon implementation had a higher final accuracy.

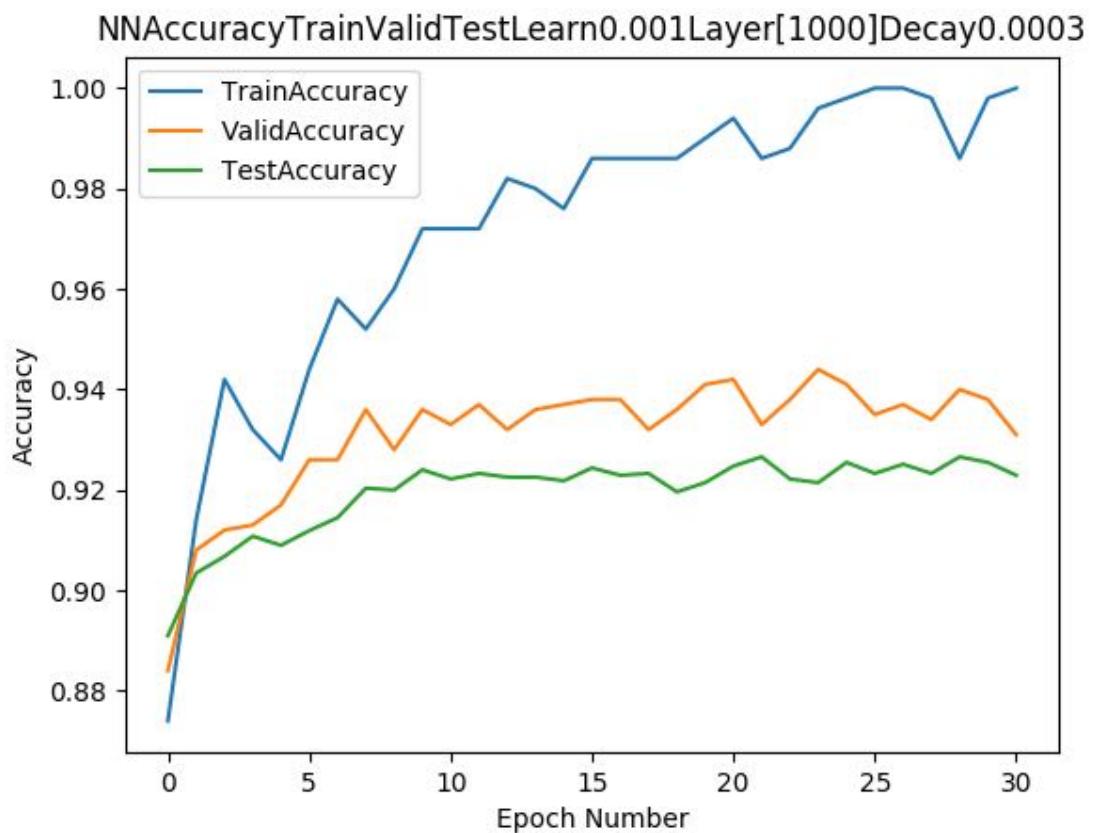


Figure 8: Training, validation and test sets cross-entropy losses and classification errors for neural network with best learning rate,  $\eta = 0.001$ .

Graphs of Best Learning Rate ( $\eta = 0.001$ )

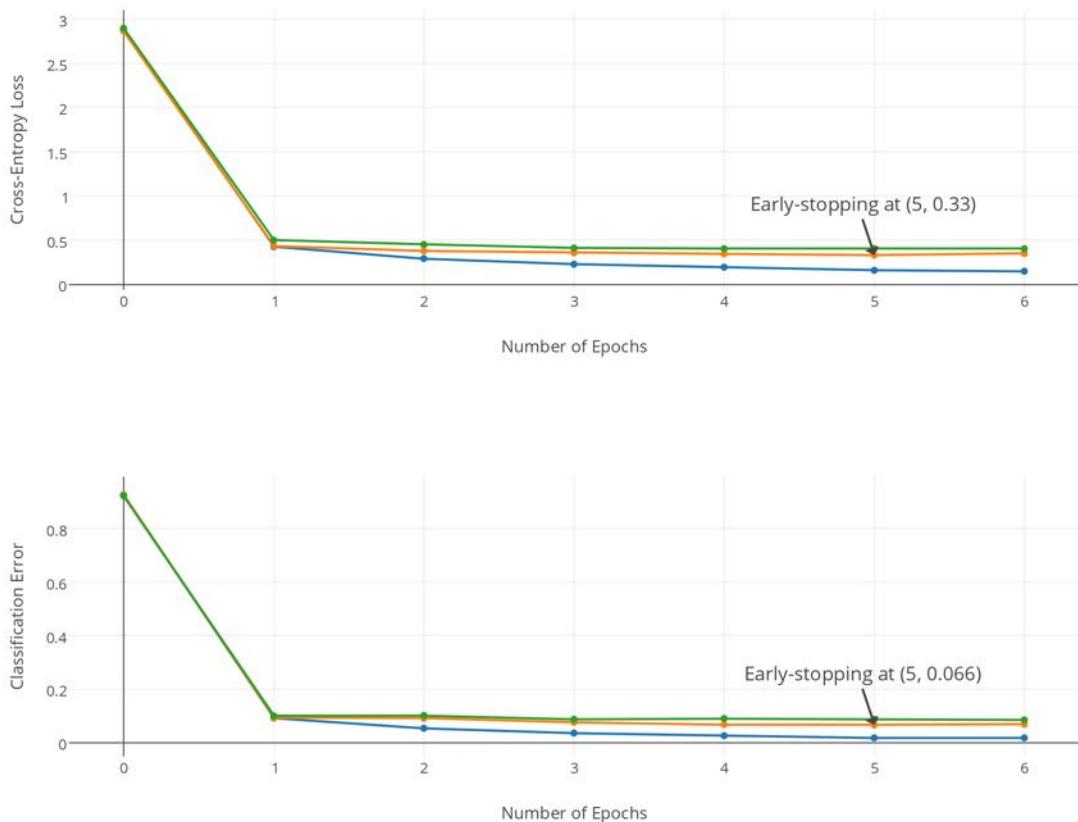


Figure 9: [Training](#), [validation](#) and [test](#) sets cross-entropy losses and classification errors for neural network with best learning rate,  $\eta = 0.001$ .

### 2.2.3 Early stopping

In our model, training convergence is implemented through early stopping. In our case, it is defined to be when the average validation loss for the latest epoch is more than of the average validation loss for the previous epoch. This is done since an increase in the average validation loss is a sign of over-fitting.

A similar early-stopping implementation on validation accuracy.

The early-stopping points are marked in Figure 9. The classification errors at early stopping are provided in Table 9 below.

Table 9: Performance of neural network at early stopping point

<i>Epoch</i>	<i>Training Error</i>	<i>Validation Error</i>	<i>Test Error</i>
20	1.2%	6.2%	8.8%

The early stopping points occur at the same epoch for both plots. The validation cross-entropy loss plot should be used to determine early-stopping, as increasing values of validation losses past the early-stopping (minimum) point is a sign of over-fitting.

## 2.3 Effect of hyperparameters

### 2.3.1 Number of hidden units

Using the best learning rate from the previous section, the number of hidden units in the neural network was adjusted *ceteris paribus* (refer to Table 10).

Table 10: Neural network hyperparameters

Hyperparameter	Value
Batch size, $B$	500
L2 regularizer, $\lambda$	$3 \times 10^{-4}$
Learning rate, $\eta$	0.001
Hidden Layers	1
Dropout Rate	0%

Table 11 summarises the performance of the models with varying hidden units. As seen from the table, having a neural network with more hidden units, so long as there is sufficient weights regularization, leads to marginally better performance.

Table 11: Performance of neural network with respect to the number of hidden units

Hidden Units	Epochs	Lowest Validation Error, %	Lowest Test Error, %
100	9	7.4	9.2
500	6	7.1	8.4
1000	6	6.1	8.1

### 2.3.2 Number of layers

The model was adjusted to have one or two hidden layers while keeping the total number of hidden units and other hyperparameters the same (refer to Table 12). The performance of these two models can be seen in Table 13.

Table 12: Neural network hyperparameters

<i>Hyperparameter</i>	<i>Value</i>
Batch size, $B$	500
L2 regularizer, $\lambda$	$3 \times 10^{-4}$
Learning rate, $\eta$	0.001
Total Hidden Units	1000
Dropout Rate	0%

Table 13: Performance of neural network with respect to number of hidden layers

<i>Hidden Layers</i>	<i>Hidden Units</i>	<i>Epochs</i>	<i>Validation Loss</i>	<i>Best Test Accuracy, %</i>
1	(1000)	6	0.349	91.8
2	(500, 500)	5	0.340	91.9

The final training error for the two-layer model turned out to be 8.4%. From the table above, the two-layer model achieves marginally higher accuracy.

The performance of the two-layer model is depicted in the figure below.

Effect of layers on model performance

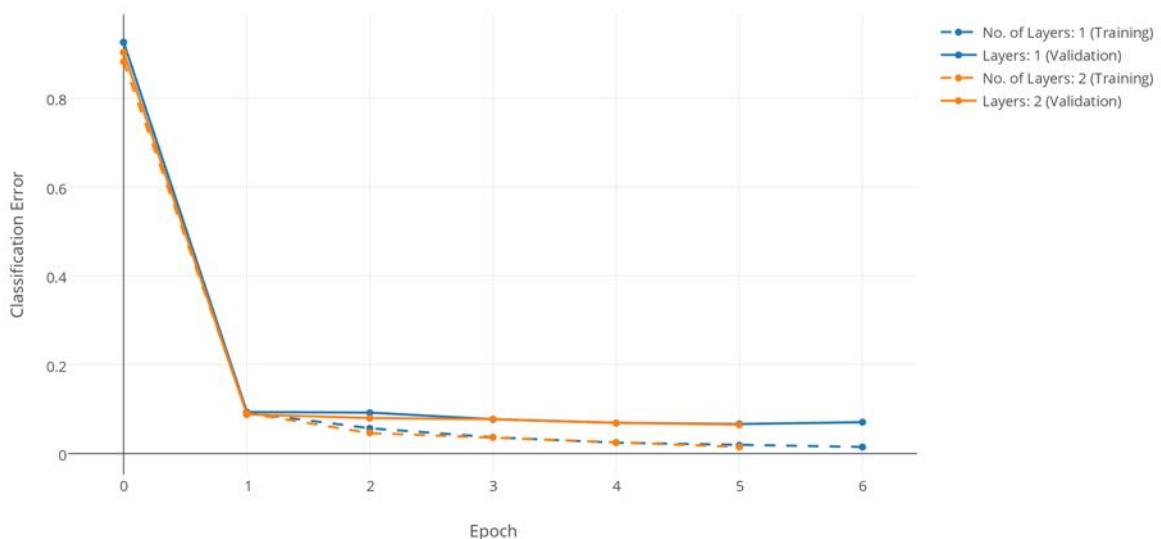


Figure 10: Training and validation classification errors for [one-layer](#) and [two-layer](#) neural network.

## 2.4 Regularization and visualization

### 2.4.1 Dropout

Dropout was introduced to the neural network at a rate of 50% (refer to Table 15). The performance of this model is shown in Figure 11.

Table 14: Neural network hyperparameters

Hyperparameter	Value
Batch size, $B$	500
L2 regularizer, $\lambda$	$3 \times 10^{-4}$
Learning rate, $\eta$	0.001
Hidden Layers	1
Hidden Units	(1000)

Table 15: Performance of neural network with respect to dropout

Dropout Rate, %	Epochs	Validation Loss	Best Test Accuracy, %
0	6	0.349	91.8
50	9	0.365	91.4

It is noted that the model with dropout takes a longer time to achieve early stopping. This makes sense as dropout forces weights to learn independently of their neighbours by turning off neighbouring activations at random. This leads to an overall result where more training is required until reach early stopping is reached.

In terms of performance, the model with dropout bears a slightly lower performance, albeit a minute one, from that without dropout. Hence, applying dropout in this situation did not improve the performance of the network but it necessarily. However, the benefits of dropout on model performance are expected to be more pronounced when applied onto a much larger or deeper neural network.

Effect of Dropout on Model Performance

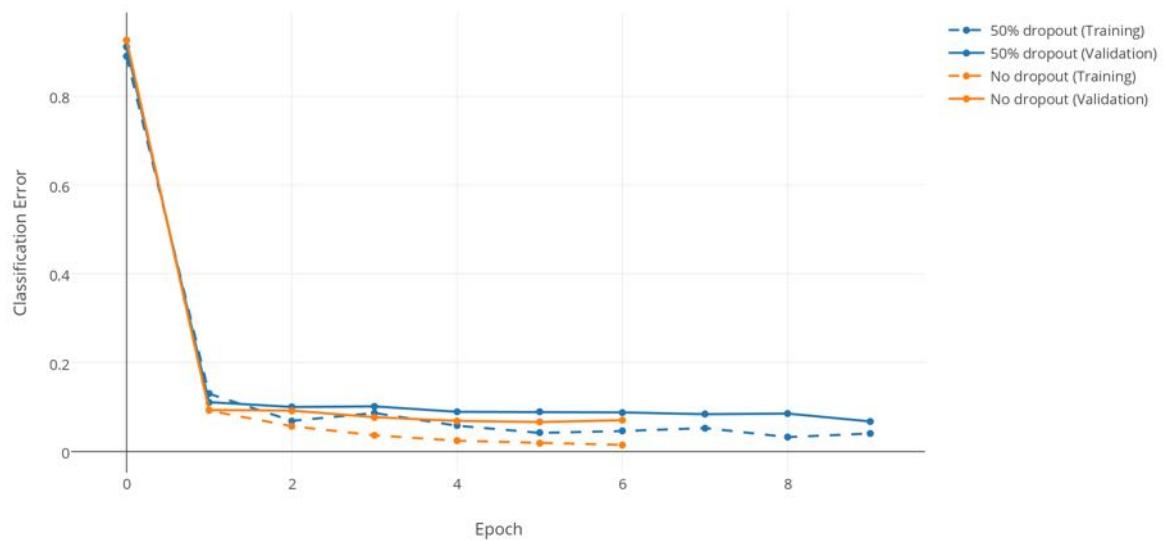


Figure 11: Training and validation classification errors [with](#) and [without](#) dropout.

## 2.4.2 Visualization

To visualise the results, the 784 incoming weights to each hidden unit is plotted as a 28 by 28 grayscale heatmap. This process is repeated to form a 40 by 25 array of heatmaps representing the 1000 hidden units.

This process is done at 25%, 50%, 75% and 100% of the early stopping points for the basic neural network (see Table 7) and the neural network with dropout (see Table 14).

From Figures 12 and 13 below, the neurons were able to pick up on patterns in the data. Upon close inspection, one can observe shades and patterns that resemble letters in the English alphabet, especially so for the weights of model without dropout.

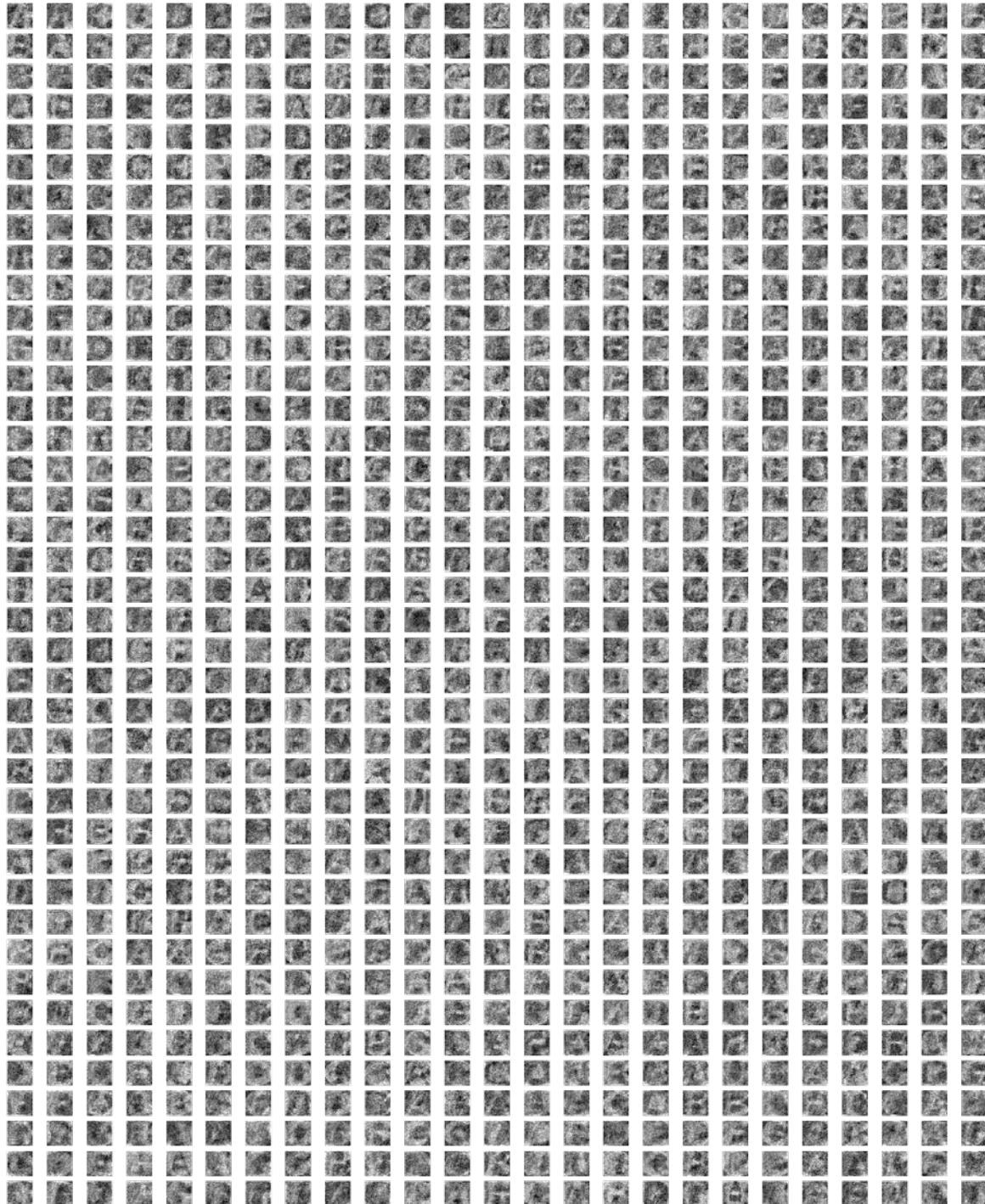
The visualised weights change very quickly from 0% (Gaussian noise) to 25% (distinguishable patterns). After that, the difference between each snapshot of the weights becomes more subtle, in tandem with the plateauing loss function of the model. This trend is observed in both visualisations involving weights with and without dropout.

One must note that the input data,  $x$ , have been mean- and variance-normalised. As a result, the patterns observed in Figure 12 is less obvious (lower contrast) than it would have been, had the input data not been normalised prior to training.

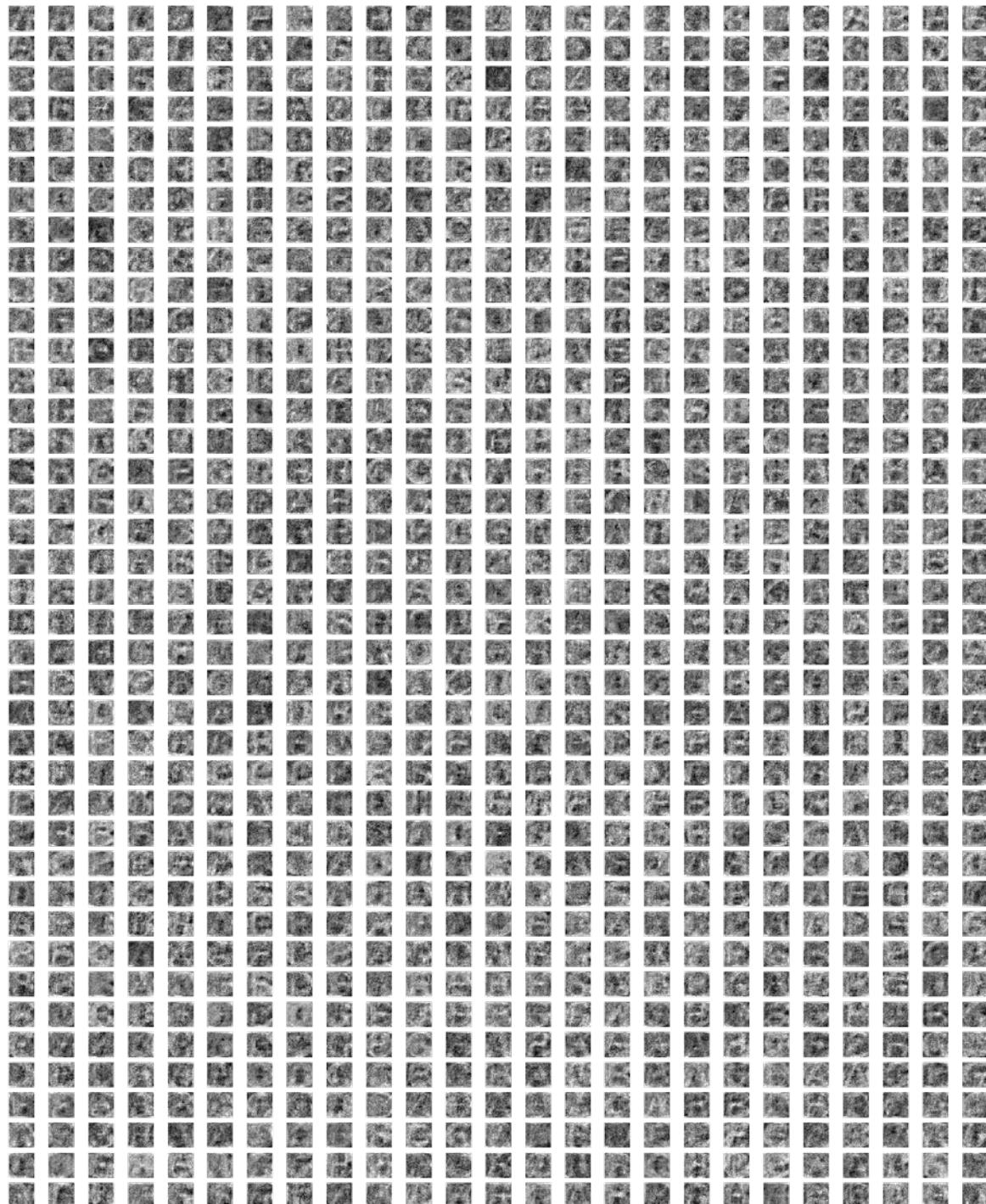
Comparing between the two sets of visualisation with and without dropout, an stark observation is that the patterns are much less distinguishable for the model with dropout. This observation can be explained by the fact that dropout out neurons stochastically trains the neighbouring neurons to learn independently of one another. Graphically, this results in each weight or cell to be to be less correlated to its neighbours, leading to less obvious patterns in the visualisations in Figure 13.

## Hidden Layer Weights (without Dropout)

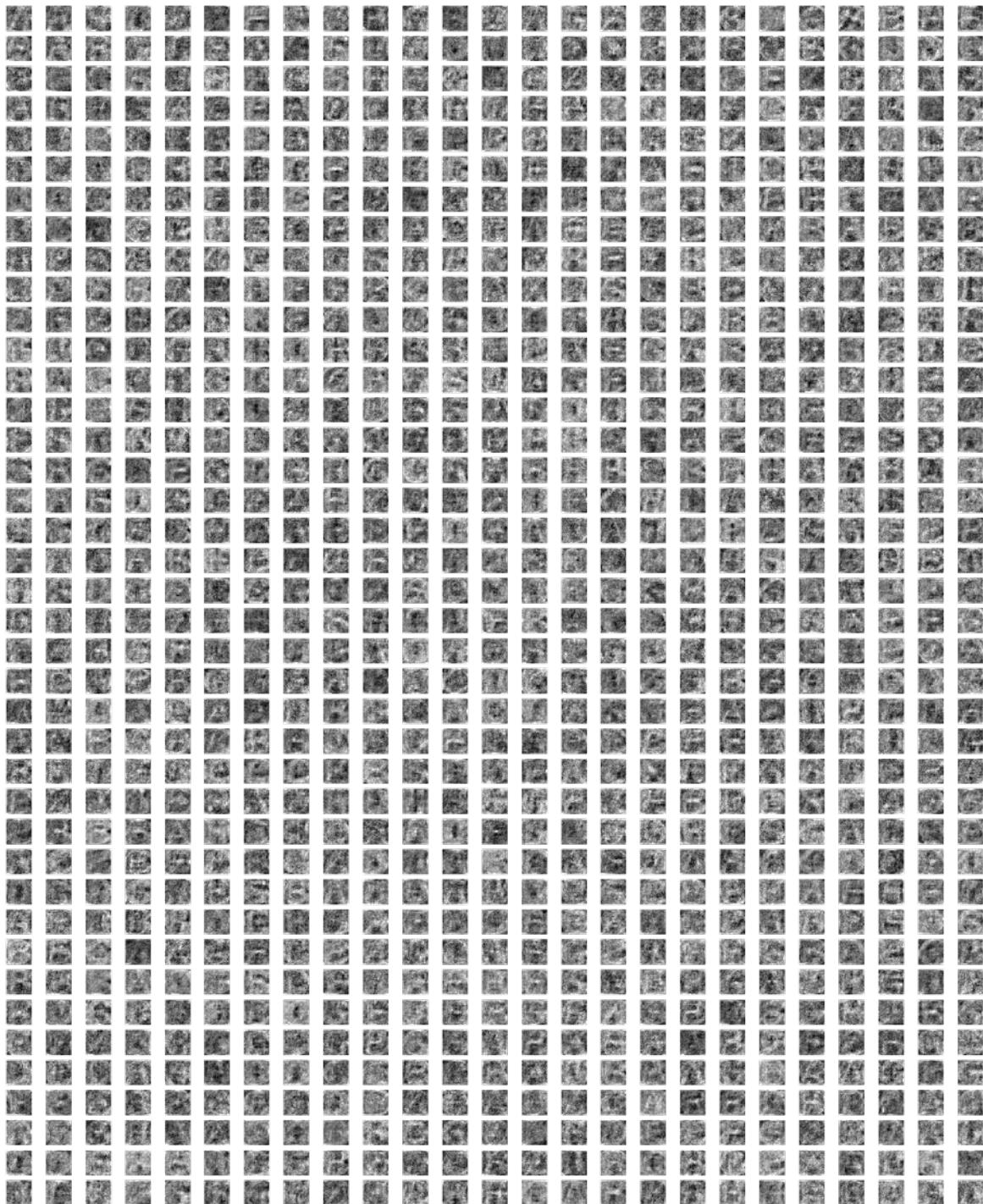
25%



50%



75%



100%

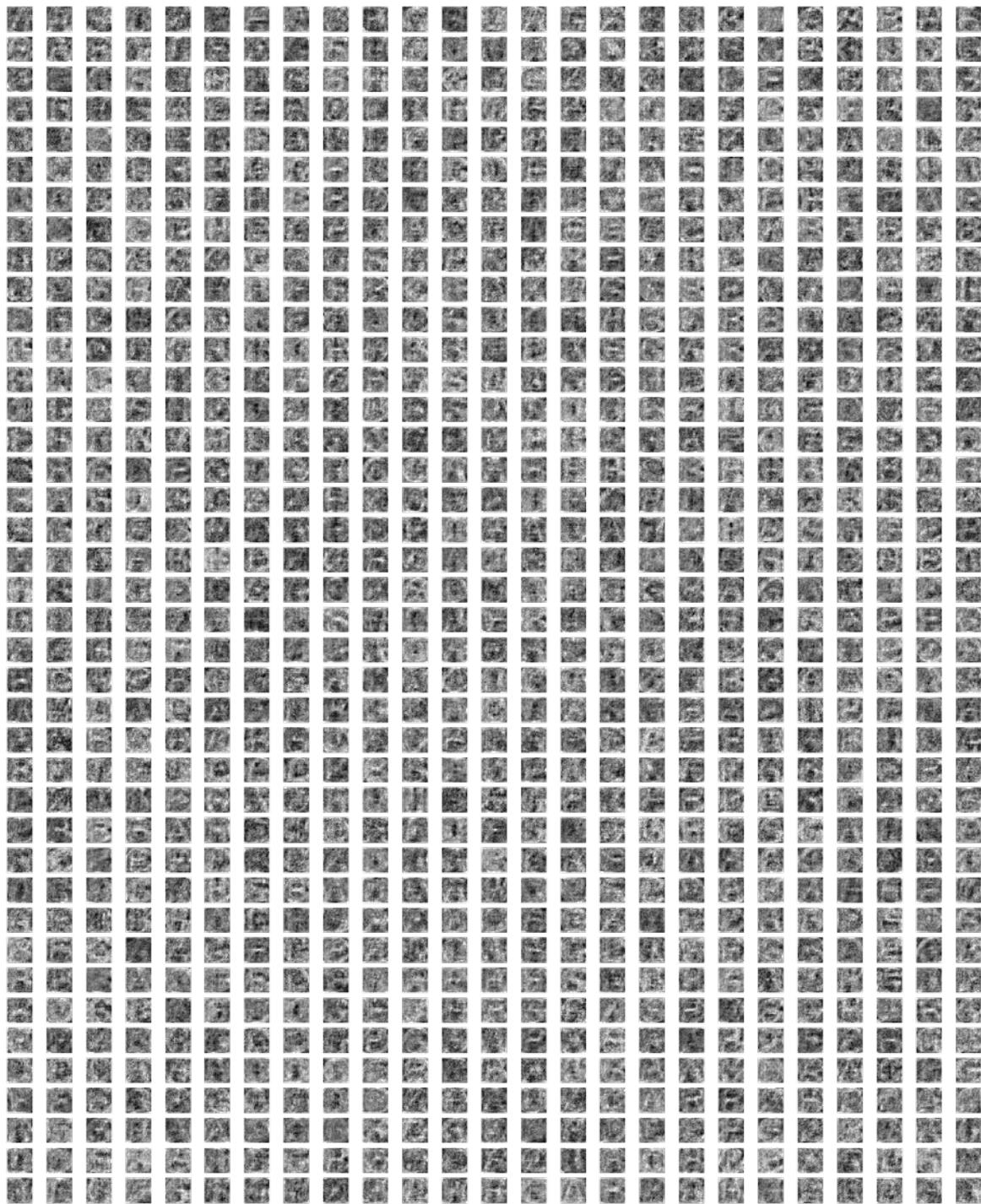
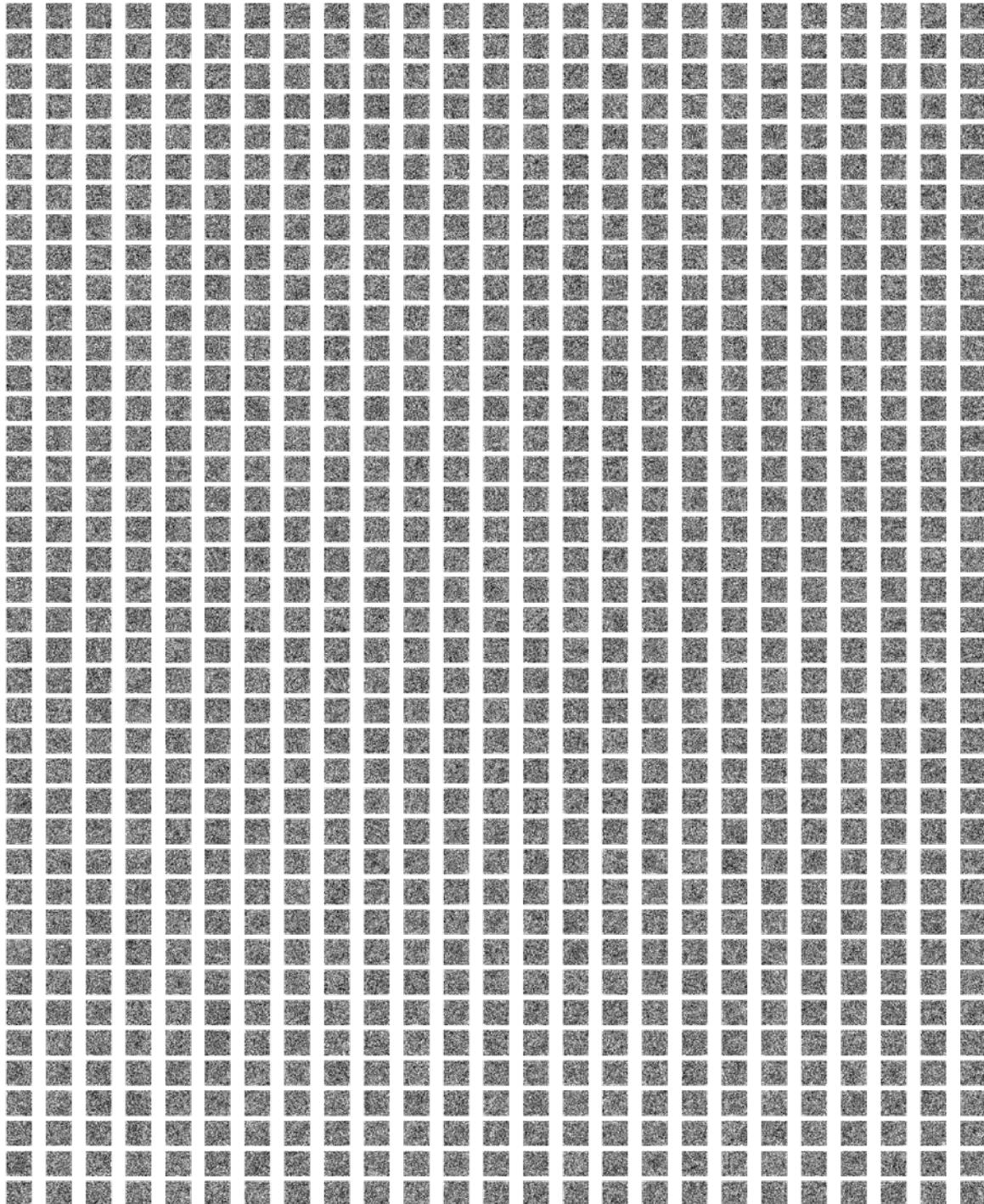


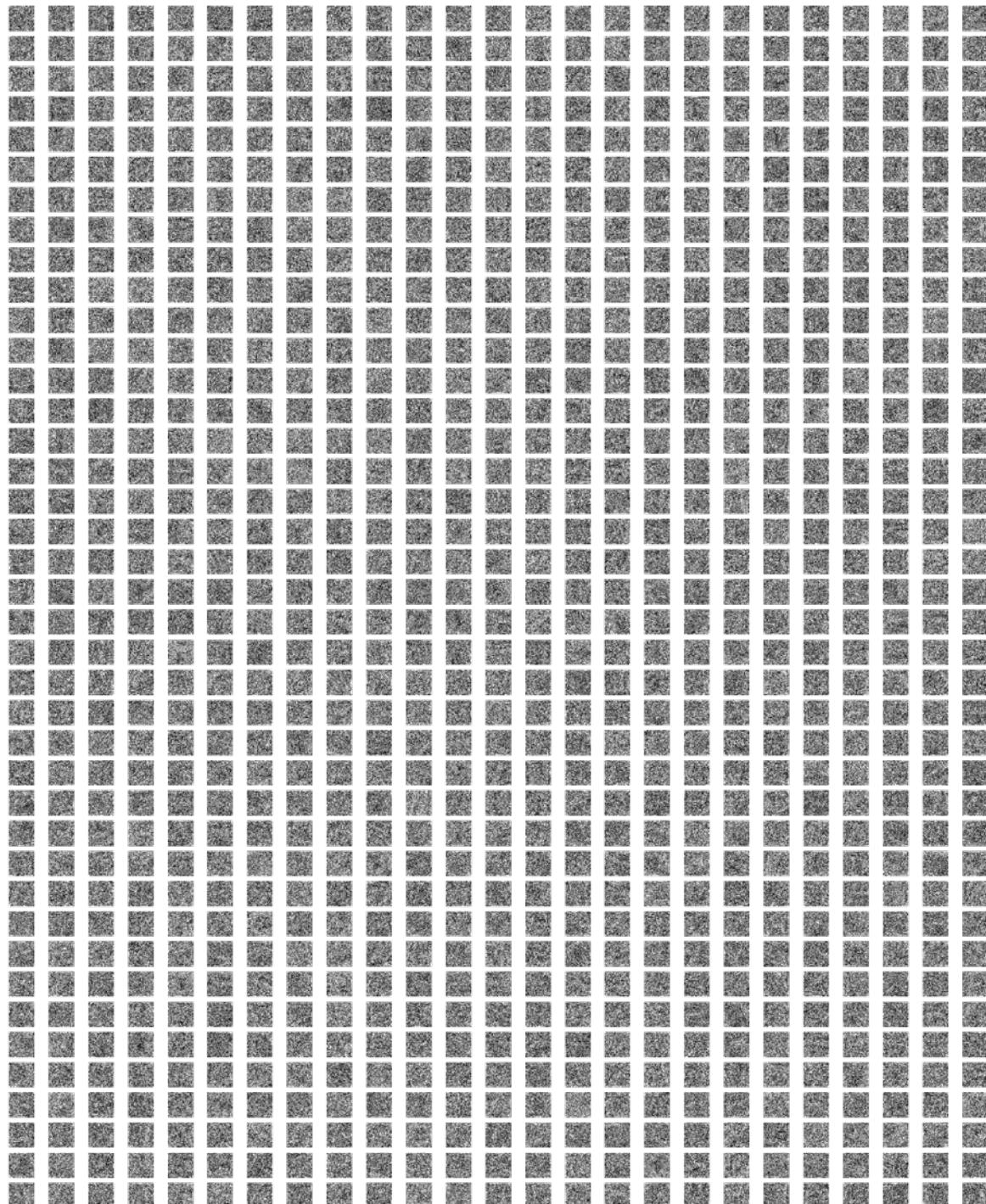
Figure 12: Visualisation of hidden layer weights for model without dropout at 25%, 50%, 75% and 100% of early stopping point.

## Hidden Layer Weights (with Dropout)

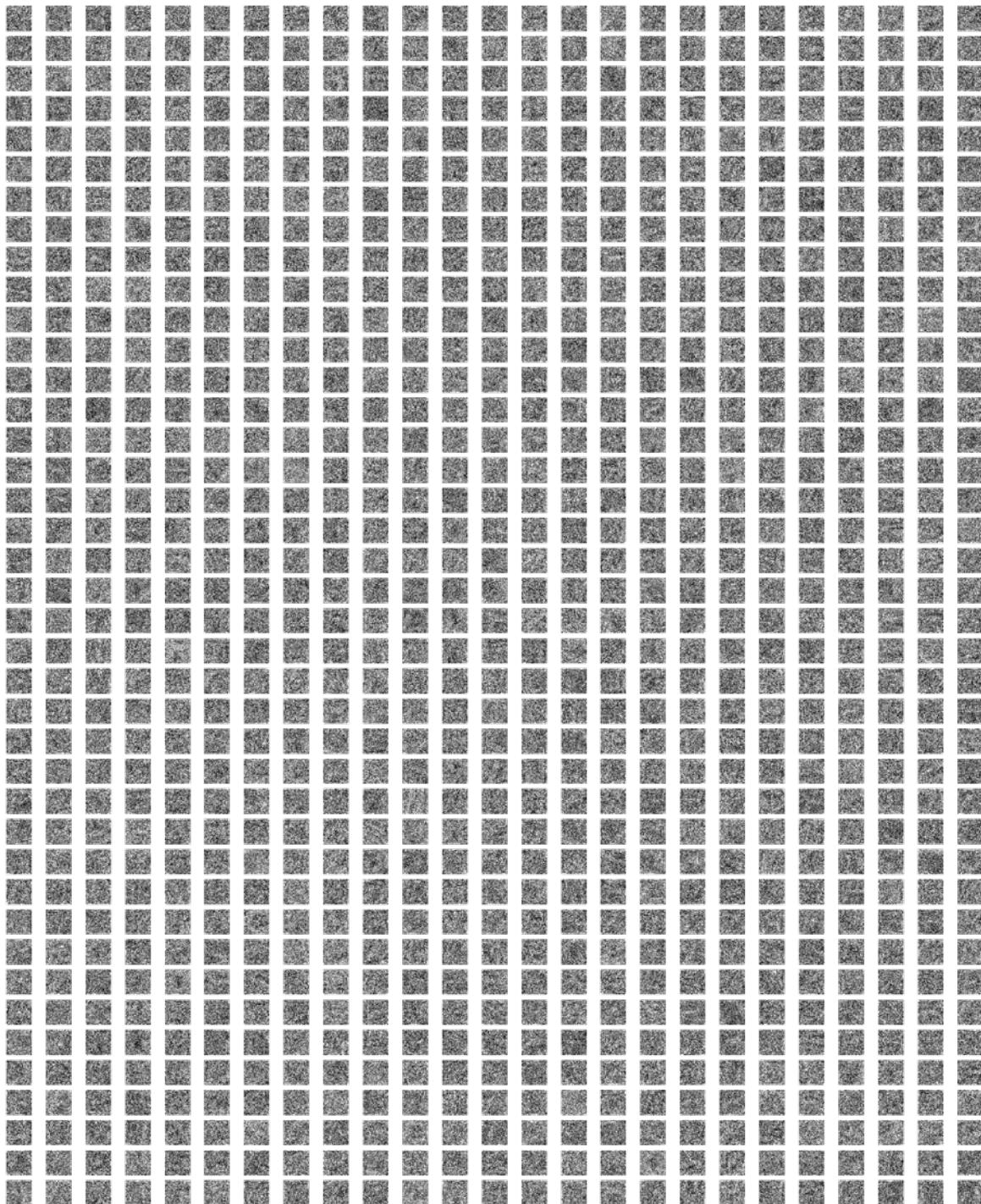
25%



50%



75%



100%

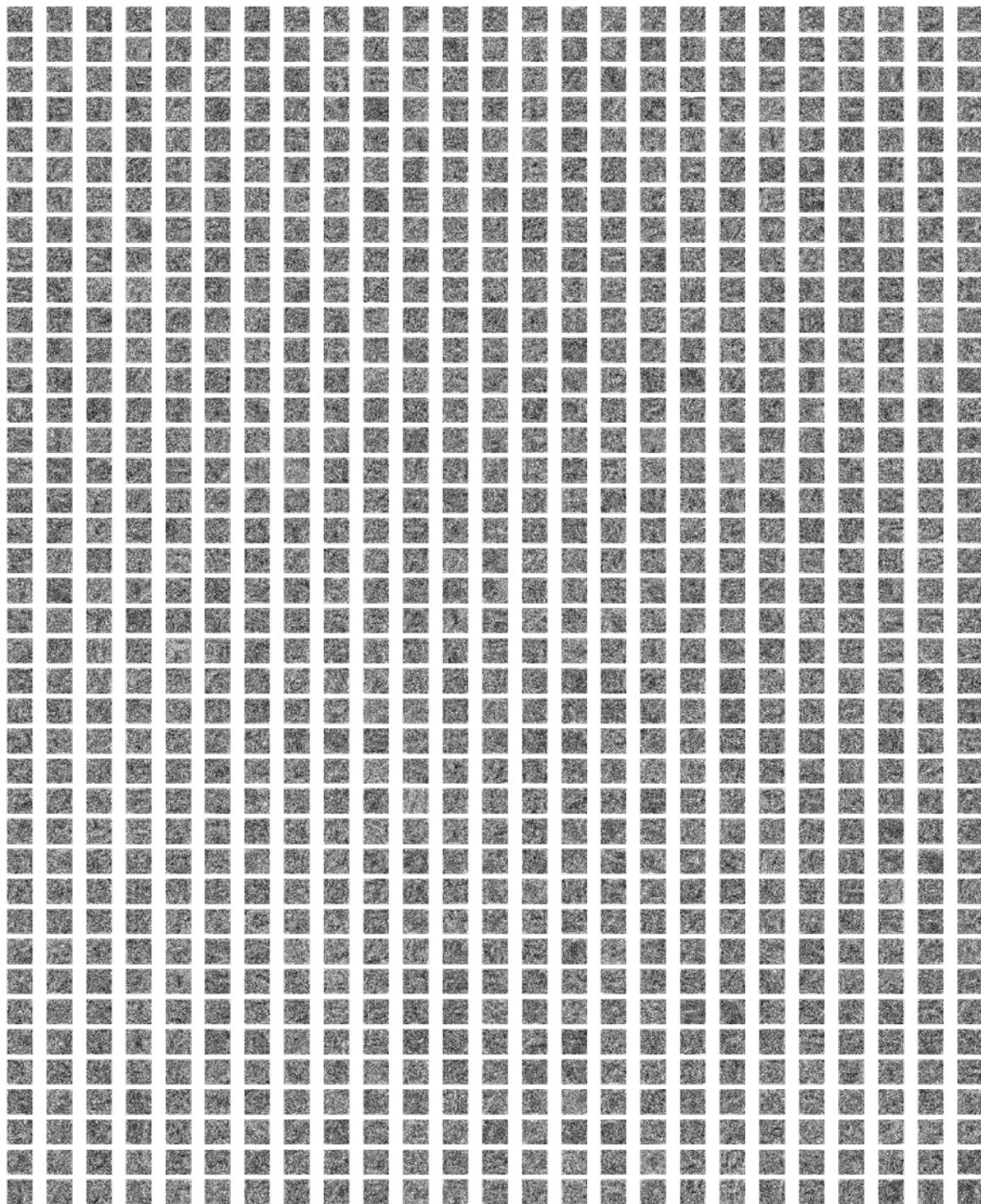


Figure 13: Visualisation of hidden layer weights for model with dropout at 25%, 50%, 75% and 100% of early stopping point.

## 2.5 Exhaustive search for the best set of hyperparameters

### 2.5.1 Random search

Neural network hyperparameters (except batch size,  $B$ ) were generated stochastically as per assignment instructions. To ensure reproducibility, the seed number '3793' was chosen to be the concatenation of the last two digits of both student numbers.

This random generation of the model hyperparameters was done five times and the results of are summarised in the table below.

Table 16: Randomly generated neural network hyperparameters and their performances

Hyperparameter	Run 1	Run 2	Run 3	Run 4	Run 5
Batch size, $B$	500	500	500	500	500
L2 regularizer, $\lambda$	1.70e-3	1.40e-3	6.38e-4	2.67e-4	5.15e-4
Learning rate, $\eta$	7.05e-4	6.85e-3	4.97e-3	3.77e-3	1.08e-3
Hidden Layers	5	4	5	2	1
Hidden Units	1	2	3	4	5
Dropout Rate	0%	50%	0%	50%	0%
Lowest Validation Error, %	5.9	11.4	7.6	9.6	7.2
Lowest Test Error, %	7.9	12.2	8.4	10.9	8.1

The results above were shared in the class Piazza webpage.

---

<sup>1</sup>(236, 438, 326, 298, 496)

<sup>2</sup>(409, 381, 103, 388)

<sup>3</sup>(480, 124, 154, 322, 361)

<sup>4</sup>(184, 377)

<sup>5</sup>(197)

## 2.5.2 Exchange ideas among the groups

From the postings for Question 2.5.2 scoured on the class Piazza webpage, our colleague *Eric Jiang* had the best performing neural network model at the time of writing. This model was generated using hyperparameter values summarised in Table 17.

We obtained comparatively similar values when we simulated his model using our random seeds. We obtained validation and test errors of 6.4% and 8.3% respectively.

Table 17: Best model<sup>6</sup> hyperparameters on Piazza and its performance

<i>Hyperparameter</i>	<i>Values</i>
Batch size, $B$	500
L2 regularizer, $\lambda$	9.10e-4
Learning rate, $\eta$	2.92e-3
Hidden Layers	5
Hidden Units	(496, 276, 486, 299, 376)
Dropout Rate	0%
Lowest Validation Error, %	7.0
Lowest Test Error, %	6.4

---

<sup>6</sup>at time of writing

### 3 Appendices

We implemented our code separately. Each figure comes from one of these two implementations.

#### 3.1 Entire Code: Chee Loong Soon's version

##### 3.1.1 Part 1

```
1 import datetime
2 import numpy as np
3 import tensorflow as tf
4 import sys
5
6 class LogisticRegression(object):
7     def __init__(self, trainData, trainTarget, validData,
8                  validTarget, testData, testTarget, numEpoch = 1,
9                  learningRate = 0.0001, weightDecay = 0.01,
10                 optimizerType = "adam", classifierType = "Binary",
11                 executeLinearRegression = False, questionTitle = ""):
12         self.trainData = trainData
13         self.trainTarget = trainTarget
14         self.validData = validData
15         self.validTarget = validTarget
16         self.testData = testData
17         self.testTarget = testTarget
18         self.miniBatchSize = 500
19         self.numEpoch = numEpoch
20         self.learningRate = learningRate
21         self.weightDecay = weightDecay
```

```

18     # To not execute Linear Regression every time
19
20     self.executeLinearRegression = executeLinearRegression
21     self.executedLinear = False
22     self.WNormalEquationSave = 0
23
24     self.accuracyNormalSave = 0
25
26
27     self.questionTitle = questionTitle
28     self.classifierType = classifierType
29     self.optimizer = 0
30
31     if optimizerType == "gd":
32
33         self.optimizer =
34             → tf.train.AdamOptimizer(learning_rate =
35             → self.learningRate)
36
37     else:
38
39         self.optimizer =
40             → tf.train.GradientDescentOptimizer(learning_rate
41             → = self.learningRate)

```

```

1 def printPlotResults(self, numUpdate, errTrain, accTrain,
2     → errValid, errTest, accValid, accTest, xAxis,
3     → yTrainErr, yValidErr, yTestErr, yTrainAcc, yValidAcc,
4     → yTestAcc, yTrainNormalAcc = 0, yValidNormalAcc = 0,
5     → yTestNormalAcc = 0):
6
7     global figureCount
8
9     import matplotlib.pyplot as plt
10
11    print self.classifierType
12
13    print "LearningRate: " , self.learningRate, " Mini
14        → batch Size: " , self.miniBatchSize
15
16    print "NumEpoch: " , self.numEpoch

```

```

7   print "Iter: ", numUpdate
8
9   print "Final Train MSE: ", errTrain
10
11  print "Final Train Acc: ", accTrain
12
13  print "Final Valid MSE: ", errValid
14
15  print "Final Test MSE: ", errTest
16
17  print "Final Valid Acc: ", accValid
18
19  print "Final Test Acc: ", accTest
20
21  paramString = "LearnRate" + str(self.learningRate) +
22      "NumEpoch" + str(self.numEpoch)
23
24  typeLossStr = "Loss"
25
26  typeAccuracyStr = "Accuracy"
27
28  iterationStr = "Iteration"
29
30  trainStr = "Train"
31
32  validStr = "Valid"
33
34  testStr = "Test"
35
36  trainLossStr = trainStr + typeLossStr
37
38  validLossStr = validStr + typeLossStr
39
40  testLossStr = testStr + typeLossStr
41
42
43  trainAccStr = trainStr + typeAccuracyStr
44
45  validAccStr = validStr + typeAccuracyStr
46
47  testAccStr = testStr + typeAccuracyStr
48
49
50  linearLabelStr = "LinearRegression"
51
52
53  figureCount = figureCount + 1
54
55  plt.figure(figureCount)
56
57  title = self.classifierType + typeLossStr + trainStr +
58      paramString
59
60  plt.title(title)

```

```

35     plt.xlabel(iterationStr)
36
37     plt.ylabel(typeLossStr)
38
39     plt.plot(np.array(xAxis), np.array(yTrainErr), label =
40             ↳ trainLossStr)
41
42     plt.legend()
43
44     plt.savefig(self.questionTitle + title + ".png")
45
46     plt.close()
47
48     plt.clf()
49
50
51     figureCount = figureCount + 1
52
53     plt.figure(figureCount)
54
55     title = self.classifierType + typeLossStr + validStr +
56             ↳ paramStr
57
58     plt.title(title)
59
60     plt.xlabel(iterationStr)
61
62     plt.ylabel(typeLossStr)
63
64     plt.plot(np.array(xAxis), np.array(yValidErr), label =
65             ↳ validLossStr)
66
67     plt.legend()
68
69     plt.savefig(self.questionTitle + title + ".png")
70
71     plt.close()
72
73     plt.clf()
74
75
76     figureCount = figureCount + 1
77
78     plt.figure(figureCount)
79
80     title = self.classifierType + typeLossStr + testStr +
81             ↳ paramStr
82
83     plt.title(title)
84
85     plt.xlabel(iterationStr)
86
87     plt.ylabel(typeLossStr)

```

```

61     plt.plot(np.array(xAxis), np.array(yTestErr), label =
62             ↳ testLossStr)
63
64     plt.legend()
65
66     plt.savefig(self.questionTitle + title + ".png")
67
68     plt.close()
69
70     plt.clf()

```

```

1     figureCount = figureCount + 1
2
3     plt.figure(figureCount)
4
5     title = self.classifierType + typeLossStr + trainStr +
6             ↳ validStr + testStr + paramStr
7
8     plt.title(title)
9
10    plt.xlabel(iterationStr)
11
12    plt.ylabel(typeLossStr)
13
14    plt.plot(np.array(xAxis), np.array(yTrainErr), label =
15             ↳ trainLossStr)
16
17    plt.plot(np.array(xAxis), np.array(yValidErr), label =
18             ↳ validLossStr)
19
20    plt.plot(np.array(xAxis), np.array(yTestErr), label =
21             ↳ testLossStr)
22
23    plt.legend()
24
25    plt.savefig(self.questionTitle + title + ".png")
26
27    plt.close()
28
29    plt.clf()

```

```

1     # Accuracies
2
3     figureCount = figureCount + 1
4
5     plt.figure(figureCount)
6
7     title = self.classifierType + typeAccuracyStr +
8             ↳ trainStr + paramStr

```

```

5      plt.title(title)
6      plt.xlabel(iterationStr)
7      plt.ylabel(typeAccuracyStr)
8      plt.plot(np.array(xAxis), np.array(yTrainAcc), label =
9          ↳ trainAccStr)
10     accLabel = trainStr + typeAccuracyStr
11
12     if self.executeLinearRegression:
13
14         plt.plot(np.array(xAxis),
15             ↳ np.array(yTrainNormalAcc), label =
16             ↳ linearLabelStr)
17
18     plt.legend()
19
20     plt.savefig(self.questionTitle + title + ".png")
21     plt.close()
22
23     plt.clf()
24
25
26     figureCount = figureCount + 1
27     plt.figure(figureCount)
28
29     title = self.classifierType + typeAccuracyStr +
30         ↳ validStr + paramStr
31
32     plt.title(title)
33     plt.xlabel(iterationStr)
34     plt.ylabel(typeAccuracyStr)
35
36     plt.plot(np.array(xAxis), np.array(yValidAcc), label =
37         ↳ validAccStr)
38
39     if self.executeLinearRegression:
40
41         plt.plot(np.array(xAxis),
42             ↳ np.array(yValidNormalAcc), label =
43             ↳ linearLabelStr)
44
45     plt.legend()
46
47     plt.savefig(self.questionTitle + title + ".png")

```

```

28     plt.close()
29
30
31     figureCount = figureCount + 1
32
33     title = self.classifierType + typeAccuracyStr +
34         ↳ testStr + paramStr
35
36     plt.figure(figureCount)
37
38     plt.title(title)
39
40     plt.xlabel(iterationStr)
41
42     plt.ylabel(typeAccuracyStr)
43
44     plt.plot(np.array(xAxis), np.array(yTestAcc), label =
45         ↳ testAccStr)
46
47     if self.executeLinearRegression:
48
49         plt.plot(np.array(xAxis),
50             ↳ np.array(yTestNormalAcc), label =
51             ↳ linearLabelStr)
52
53
54     plt.legend()
55
56     plt.savefig(self.questionTitle + title + ".png")
57
58     plt.close()
59
60     plt.clf()
61
62
63
64
65     figureCount = figureCount + 1
66
67     title = self.classifierType + typeAccuracyStr +
68         ↳ trainStr + validStr + testStr + paramStr
69
70     plt.figure(figureCount)
71
72     plt.title(title)
73
74     plt.xlabel(iterationStr)
75
76     plt.ylabel(typeAccuracyStr)
77
78     plt.plot(np.array(xAxis), np.array(yTrainAcc), label =
79         ↳ trainAccStr)
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

```

53     plt.plot(np.array(xAxis), np.array(yValidAcc), label =
54         ↪ validAccStr)
55     plt.plot(np.array(xAxis), np.array(yTestAcc), label =
56         ↪ testAccStr)
57     plt.legend()
58     plt.savefig(self.questionTitle + title + ".png")
59     plt.close()
60     plt.clf()
61
62
63     print self.questionTitle + self.classifierType
64     print "Max Test Accuracy is: ",
65         ↪ max(np.array(yTestAcc))

```

```

1      # Logistic Regression
2      def LogisticRegressionMethodBinary(self):
3          """
4              Implements logistic regression and cross-entropy loss.
5              using tf.nn.sigmoid_cross_entropy_with_logits
6              Weight-decay coefficient = 0.01
7              Mini-batch size = B = 500
8              Two Class notMNIST dataset
9              Output:
10                 Plots best training curve
11                     Cross-entropy Loss vs Number of updates
12                     Classification Accuracy vs Number of Updates
13                 Plots best test curve
14                     Cross-entropy Loss vs Number of updates
15                     Classification Accuracy vs Number of Updates
16                     Best Test Classification Accuracy

```

```

17      """
18
19      maxTestClassificationAccuracy = 0.0
20      W, b, X, y_target, y_predicted, crossEntropyError,
21          ↳ train , needTrain, accuracy, WNormalEquation,
22          ↳ accuracyNormal, Xall, y_targetAll =
23          ↳ self.buildGraphBinary()
24
25      # Session
26
27      init = tf.global_variables_initializer()
28      sess = tf.InteractiveSession()
29
30      sess.run(init)
31
32      initialW = sess.run(W)
33
34      initialb = sess.run(b)
35
36      currEpoch = 0
37
38      wList = []
39
40      xAxis = []
41
42      yTrainErr = []
43
44      yValidErr = []
45
46      yTestErr = []
47
48      yTrainAcc = []
49
50      yValidAcc = []
51
52      yTestAcc = []
53
54      yTrainNormalAcc = []
55
56      yValidNormalAcc = []
57
58      yTestNormalAcc = []
59
60      numUpdate = 0
61
62      step = 0
63
64      errTrain = -1
65
66      errValid = -1

```

```

44     errTest = -1
45     accTrain = -1
46     accValid = -1
47     accTest = -1
48
while currEpoch < self.numEpoch:
49
    self.trainData, self.trainTarget =
        ↳ self.ShuffleBatches(self.trainData,
        ↳ self.trainTarget)
50
    step = 0
51
while step*self.miniBatchSize <
        ↳ self.trainData.shape[0]:
52
        # train comes from BuildGraph's optimization
        ↳ method
53
        # returnedValues =
        ↳ sess.run ([whatYouWantToReturnThatWereReturnedFromBuildGraph])
54
        #
        ↳ feed_dict{valuesToFeedIntoPlaceHoldersThatWereReturnedFromBuildGraph}
55
        # sess.run() executes whatever graph you built
        ↳ once up to the point where it needs to
        ↳ fetch
56
        # and fetches everything that's in
        ↳ ([variablesToFetch])
57
        # Thus, if you don't fetch 'train =
        ↳ optimizer.minimize(loss)', it won't
        ↳ optimize it
58
        _, errTrain, currentW, currentb, yhat,
        ↳ accTrain, currentWNORMALEquation,
        ↳ accTrainNormal= sess.run([train,
        ↳ crossEntropyError, W, b, y_predicted,
        ↳ accuracy, WNORMALEquation,
        ↳ accuracyNormal], feed_dict={X:
        ↳ np.reshape(self.trainData[step*self.miniBatchSize:(step+1)*self.miniBatchSize, :],
        ↳ (self.miniBatchSize, 784)), Xall:
        ↳ np.reshape(self.trainData, (3500, 784)),
        ↳ y_targetAll: self.trainTarget, y_target:

```

```

59         wList.append(currentW)
60
61         step = step + 1
62
63         xAxis.append(numUpdate)
64
65         numUpdate += 1
66
67         yTrainErr.append(errTrain)
68
69         yTrainAcc.append(accTrain)
70         yTrainNormalAcc.append(accTrainNormal)

71
72         # These will not optimize the function cause
73         # you did not fetch 'train'
74         # So it won't have to execute that.

75         errValid, accValid, accValidNormal =
76
77             ↳ sess.run([crossEntropyError, accuracy,
78             ↳ accuracyNormal], feed_dict={X:
79                 ↳ np.reshape(self.validData,
80                 ↳ (self.validData.shape[0], 784)), Xall:
81                 ↳ np.reshape(self.trainData, (3500, 784)),
82                 ↳ y_targetAll: self.trainTarget, y_target:
83                 ↳ self.validTarget, needTrain: False})

```

```

1           errTest, accTest, accTestNormal =
2             ↳ sess.run([crossEntropyError, accuracy,
3                         ↳ accuracyNormal], feed_dict={X:
4                           ↳ np.reshape(self testData,
5                             ↳ (self testData.shape[0], 784)), Xall:
6                               ↳ np.reshape(self trainData, (3500, 784)),
7                                 ↳ y_targetAll: self trainTarget, y_target:
8                                   ↳ self testTarget, needTrain: False})
9
10          yValidErr.append(errValid)
11          yTestErr.append(errTest)
12          yValidAcc.append(accValid)
13          yValidNormalAcc.append(accValidNormal)
14          yTestAcc.append(accTest)
15          yTestNormalAcc.append(accTestNormal)
16
17          currEpoch += 1
18
19          logStdOut("e" + str(currEpoch))
20
21          self.printPlotResults(numUpdate, errTrain, accTrain,
22            ↳ errValid, errTest, accValid, accTest, xAxis,
23              ↳ yTrainErr, yValidErr, yTestErr, yTrainAcc,
24                ↳ yValidAcc, yTestAcc, yTrainNormalAcc,
25                  ↳ yValidNormalAcc, yTestNormalAcc)

```

```

1 def LogisticRegressionMethodMulti(self):
2     """
3         Implements logistic regression and cross-entropy loss.
4         using tf.nn.sigmoid_cross_entropy_with_logits
5         Weight-decay coefficient = 0.01
6         Mini-batch size = B = 500
7         Two Class notMNIST dataset

```

```

8   Output:
9     Plots best training curve
10    Cross-entropy Loss vs Number of updates
11    Classification Accuracy vs Number of Updates
12    Plots best test curve
13    Cross-entropy Loss vs Number of updates
14    Classification Accuracy vs Number of Updates
15    Best Test Classification Accuracy
16    """
17
18    maxTestClassificationAccuracy = 0.0
19    W, b, X, y_target, y_predicted, crossEntropyError,
20      ↳ train , needTrain, accuracy =
21      ↳ self.buildGraphMulti()
22
23    # Session
24    init = tf.global_variables_initializer()
25    sess = tf.InteractiveSession()
26    sess.run(init)
27    initialW = sess.run(W)
28    initialb = sess.run(b)
29    currEpoch = 0
30    wList = []
31    xAxis = []
32    yTrainErr = []
33    yValidErr = []
34    yTestErr = []
35    yTrainAcc = []
36    yValidAcc = []
37    yTestAcc = []

```



```

55         wList.append(currentW)
56
57         step = step + 1
58
59         xAxis.append(numUpdate)
60
61         numUpdate += 1
62
63         yTrainErr.append(errTrain)
64
65         yTrainAcc.append(accTrain)
66
67         # These will not optimize the function cause
68         # you did not fetch 'train'
69
70         # So it won't have to execute that.
71
72         errValid, accValid =
73
74             ↳ sess.run([crossEntropyError, accuracy],
75             ↳ feed_dict={X: np.reshape(self.validData,
76             ↳ (self.validData.shape[0], 784)), y_target:
77             ↳ self.validTarget, needTrain: False})
78
79
80         errTest, accTest =
81
82             ↳ sess.run([crossEntropyError, accuracy],
83             ↳ feed_dict={X: np.reshape(self.testData,
84             ↳ (self.testData.shape[0], 784)), y_target:
85             ↳ self.testTarget, needTrain: False})
86
87
88         yValidErr.append(errValid)
89
90         yTestErr.append(errTest)
91
92         yValidAcc.append(accValid)
93
94         yTestAcc.append(accTest)
95
96         currEpoch += 1
97
98         logStdOut("e" + str(currEpoch))
99
100        self.printPlotResults(numUpdate, errTrain, accTrain,
101                           ↳ errValid, errTest, accValid, accTest, xAxis,
102                           ↳ yTrainErr, yValidErr, yTestErr, yTrainAcc,
103                           ↳ yValidAcc, yTestAcc)

```

```

1   def buildGraphMulti(self) :
2       # Parameters to train
3       # Images are 28*28 = 784 pixels
4
5       # Multi-class Clasification (10 classes)
6       # note: very important to increase dimensions W & b to
7           ↪ train each
8       # separately using cross entropy error
9       W = tf.Variable(tf.truncated_normal(shape=[784, 10],
10                                         ↪ stddev=0.5), name='weights')
11      b = tf.Variable(tf.zeros([10]), name='biases')
12
13
14      # Supervised Inputs
15      X = tf.placeholder(tf.float32, [None, 784],
16                          ↪ name='input_x')
17
18      # y_target for binary class classification
19      y_target = tf.placeholder(tf.float32, [None, 10],
20                                ↪ name='target_y')
21
22      # Label to know if it should train or simply return
23          ↪ the errors
24      needTrain = tf.placeholder(tf.bool)
25
26
27      weightDecayCoeff =
28          ↪ tf.div(tf.constant(self.weightDecay),tf.constant(2.0))
29
30
31      # Graph definition
32      y_predicted = tf.matmul(X, W) + b
33
34

```

```

25      # Multi-class Classification
26
27      correctPred = tf.equal(tf.argmax(y_predicted, 1),
28                            tf.argmax(y_target, 1))
29
30      accuracy = tf.reduce_mean(tf.cast(correctPred,
31                                     "float"))
32
33      # Weight Decay Error calculation
34
35      weightDecayMeanSquareError =
36          tf.reduce_mean(tf.square(W))
37
38      weightDecayError = tf.multiply(weightDecayCoeff,
39                                     weightDecayMeanSquareError)
40
41
42      # Cross Entropy Softmax Error Multi-class
43
44      # note: Cross entropy only works with values from 0 to
45      #       1, so multi-class must be one hot encoded
46
47      crossEntropySoftmaxError =
48          tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
49                         logits=logits,
50                         labels=y_predicted))
51
52      crossEntropySoftmaxError =
53          tf.add(crossEntropySoftmaxError, weightDecayError)
54
55
56      # Don't train if don't have to for validation and test
57      # set
58
59      #finalTrainingError = tf.select(needTrain,
60                                     meanSquaredError, tf.constant(0.0))
61
62      #finalTrainingError = tf.select(needTrain,
63                                     crossEntropySigmoidError, tf.constant(0.0))
64
65      finalTrainingError = tf.where(needTrain,
66                                    crossEntropySoftmaxError, tf.constant(0.0))

```

```

42
43     # Training mechanism
44
45     #gdOptimizer =
46         ↳ tf.train.GradientDescentOptimizer(learning_rate =
47             ↳ self.learningRate)
48
49     adamOptimizer = tf.train.AdamOptimizer(learning_rate =
50         ↳ self.learningRate)
51
52
53     # Train and update the parameters defined
54
55     # sess.run(train) will execute the optimized function
56
57     train =
58
59         ↳ adamOptimizer.minimize(loss=finalTrainingError)
60
61
62     return W, b, X, y_target, y_predicted,
63
64         ↳ crossEntropySoftmaxError, train, needTrain,
65
66         ↳ accuracy

```

```

1     # Build the computational graph
2
3     def buildGraphBinary(self):
4
5         # Parameters to train
6
7         # Images are 28*28 = 784 pixels
8
9
10        # TODO: Update the W and b initialization so it
11            ↳ depends on inputTarget size instead of 784
12
13        # Binary Classification
14
15        W = tf.Variable(tf.truncated_normal(shape=[784, 1],
16            ↳ stddev=0.5), name='weights')
17
18        b = tf.Variable(0.0, name='biases')
19
20        # Supervised Inputs

```

```

11   X = tf.placeholder(tf.float32, [None, 784],
12                      ↳ name='input_x')
13
14   # y_target for binary class classification
15   y_target = tf.placeholder(tf.float32, [None, 1],
16                           ↳ name='target_y')
17
18   # Need all for 1.1.3
19   Xall = tf.placeholder(tf.float32, [None, 784],
20                         ↳ name='input_x')
21   y_targetAll = tf.placeholder(tf.float32, [None, 1],
22                               ↳ name='target_y')
23
24   # Label to know if it should train or simply return
25   ↳ the errors
26   needTrain = tf.placeholder(tf.bool)
27
28   weightDecayCoeff =
29         ↳ tf.div(tf.constant(self.weightDecay),tf.constant(2.0))
30
31   # Graph definition
32   y_predicted = tf.matmul(X, W) + b
33
34
35   # This works only for Binary Case
36   correctPred =
37     ↳ tf.equal(tf.cast(tf.greater_equal(y_predicted,
38                       ↳ 0.5), tf.float32), tf.floor(y_target))
39   accuracy = tf.reduce_mean(tf.cast(correctPred,
40                                "float"))
41
42
43   # Weight Decay Error calculation

```

```

32     weightDecayMeanSquareError =
33         ↪ tf.reduce_mean(tf.square(W))
34
35     weightDecayError = tf.multiply(weightDecayCoeff,
36                                     ↪ weightDecayMeanSquareError)
37
38     """
39
40     # Mean Square Error Calculation
41     # Divide by 2M instead of M
42     meanSquaredError =
43         ↪ tf.div(tf.reduce_mean(tf.reduce_mean(tf.square(y_predicted
44                                     ↪ - y_target)),
45                                     ↪ reduction_indices=1,
46                                     ↪ name='squared_error'),
47                                     ↪ name='mean_squared_error'), tf.constant(2.0))
48
49     meanSquaredError = tf.add(meanSquaredError,
50                               ↪ weightDecayError)
51
52     """
53
54
55     # Cross Entropy Sigmoid Error Binary-class Calculation
56     crossEntropySigmoidError =
57         ↪ tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=
58                         ↪ = y_predicted, labels = y_target)))
59
60     crossEntropySigmoidError =
61         ↪ tf.add(crossEntropySigmoidError, weightDecayError)
62
63
64     # Don't train if don't have to for validation and test
65         ↪ set

```

```

50
51      # For 1.1.3 to be updated to normal equation instead
52      # This was for your own gradient descent for MSE that
53      # was not needed in this assignment
54
55      #finalTrainingError = tf.where(needTrain,
56      #    ↪ meanSquaredError, tf.constant(0.0))
57
58
59      finalTrainingError = tf.where(needTrain,
60      #    ↪ crossEntropySigmoidError, tf.constant(0.0))
61
62
63      # Training mechanism
64
65      # Train and update the parameters defined
66      # sess.run(train) will execute the optimized function
67
68      train =
69      #    ↪ self.optimizer.minimize(loss=finalTrainingError)
70
71
72      # Return both errors for plotting for 1.1.3 and set
73      # weight to 0 for LogisticRegression for plotting
74      # when comparing
75
76      WNormalEquation = W
77
78      accuracyNormal = accuracy
79
80      if self.executeLinearRegression:
81          # 1.1.3 Calculate Linear Regression using Normal
82          # Equation (analytical solution)
83
84          # Concatenate 1 to account for Bias
85
86          if self.executedLinear:
87              WNormalEquation = self.WNormalEquationSave
88
89              accuracyNormal = self.accuracyNormalSave
90
91
92

```

```

73     else:
74         OnesForX =
75             ↳ tf.ones(shape=tf.pack([tf.shape(Xall)[0],
76             ↳ 1]))
77             Xnormal = tf.concat(1,[Xall , OnesForX])
78             WNormalEquation =
79                 ↳ tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(tf.tr
80                 ↳ Xnormal)), tf.transpose(Xnormal)),
81                 ↳ y_targetAll)
82             OnesForCurrX =
83                 ↳ tf.ones(shape=tf.pack([tf.shape(X)[0],
84                 ↳ 1]))
85             currX = tf.concat(1,[X , OnesForCurrX])
86             y_predictedNormal = tf.matmul(currX,
87                 ↳ WNormalEquation)
88             correctPredNormal =
89                 ↳ tf.equal(tf.cast(tf.greater_equal(y_predictedNormal,
90                 ↳ 0.5), tf.float32), tf.floor(y_target))
91             accuracyNormal =
92                 ↳ tf.reduce_mean(tf.cast(correctPredNormal,
93                 ↳ "float"))
94             self.WNormalEquationSave = WNormalEquation
95             self.accuracyNormalSave = accuracyNormal
96             self.executedLinear = True
97
98
99     return W, b, X, y_target, y_predicted,
100    ↳ crossEntropySigmoidError, train, needTrain,
101    ↳ accuracy, WNormalEquation, accuracyNormal, Xall,
102    ↳ y_targetAll

```

```

1  def ShuffleBatches (self, trainData, trainTarget):
2      # Gets the state as the current time
3      rngState = np.random.get_state()
4      np.random.shuffle(trainData)
5      np.random.set_state(rngState)
6      np.random.shuffle(trainTarget)
7      return trainData, trainTarget

```

```

1  #-----
2  def convertOneHot (targetValues):
3      numClasses = np.max(targetValues) + 1
4      return np.eye(numClasses) [targetValues]

```

```

1  def ExecuteBinary (questionTitle, numEpoch, learningRates,
2      → weightDecay, optimizerType, executeLinearRegression):
3      classifierType = "Binary"
4      print questionTitle + classifierType
5      for learningRate in learningRates:
6          with np.load("notMNIST.npz") as data :
7              # Need to get new data from scratch everything for
8              → a new learning rate
9              Data, Target = data ["images"], data ["labels"]
10             posClass = 2
11             negClass = 9
12             dataIndx = (Target==posClass) + (Target==negClass)
13             Data = Data[dataIndx]/255.
14             Target = Target[dataIndx].reshape(-1, 1)
15             Target [Target==posClass] = 1
16             Target [Target==negClass] = 0

```

```

15     np.random.seed(521)
16
17     randIndx = np.arange(len(Data))
18
19     np.random.shuffle(randIndx)
20
21     Data, Target = Data[randIndx], Target[randIndx]
22
23     trainData, trainTarget = Data[:3500],
24
25         ↳ Target[:3500]
26
27     validData, validTarget = Data[3500:3600],
28
29         ↳ Target[3500:3600]
30
31     testData, testTarget = Data[3600:], Target[3600:]
32
33     # Binary Class Linear Regression = 0.001
34
35     # Binary Class Logistic Regression Sigmoid = 0.01
36
37     # Cross entropy loss only works from 0 to 1! Not
38
39         ↳ from 0 to 9 for multi-class
40
41     tf.reset_default_graph()
42
43     l = LogisticRegression(trainData, trainTarget,
44
45         ↳ validData, validTarget, testData, testTarget,
46
47         ↳ numEpoch, learningRate, weightDecay,
48
49         ↳ optimizerType, classifierType,
50
51         ↳ executeLinearRegression, questionTitle)
52
53     l.LogisticRegressionMethodBinary()
54
55     logElapsedTime(questionTitle + classifierType +
56
57         ↳ str(learningRate))

```

```

1 def ExecuteMulti(questionTitle, numEpoch, learningRates,
2
2     ↳ weightDecay, optimizerType, executeLinearRegression):
3
3     classifierType = "Multi"
4
4     print questionTitle + classifierType
5
5     with np.load("notMNIST.npz") as data:
6
6         Data, Target = data["images"], data["labels"]

```

```

6   for learningRate in learningRates:
7       np.random.seed(521)
8       randIndx = np.arange(len(Data))
9       np.random.shuffle(randIndx)
10      Data = Data[randIndx]/255.
11      Target = Target[randIndx]
12      trainData, trainTarget = Data[:15000],
13          ↳ Target[:15000]
14      validData, validTarget = Data[15000:16000],
15          ↳ Target[15000:16000]
16      # Target values are from 0 to 9
17      testData, testTarget = Data[16000:],
18          ↳ Target[16000:]
19      trainTarget = convertOneHot(trainTarget)
20      validTarget = convertOneHot(validTarget)
21      testTarget = convertOneHot(testTarget)
22      # Multiclass Classification Logistic Regression
23          ↳ Softmax = 0.01
24      tf.reset_default_graph()
25      l = LogisticRegression(trainData, trainTarget,
26          ↳ validData, validTarget, testData, testTarget,
27          ↳ numEpoch, learningRate, weightDecay,
28          ↳ optimizerType, classifierType,
29          ↳ executeLinearRegression, questionTitle)
30      l.LogisticRegressionMethodMulti()
31      logElapsed Time(questionTitle + classifierType +
32          ↳ str(learningRate))

```

```

1 # Global for logging
2 questionTitle = "" # Need to be global for logging to work
3 startTime = datetime.datetime.now()
4 figureCount = 1 # To not overwrite existing pictures
5
6 def logStdOut(message):
7     # Temporary print to std out
8     sys.stdout = sys.__stdout__
9     print message
10    # Continue editing same file
11    sys.stdout = open("result" + questionTitle + ".txt", "a")
12
13 def logElapsedTime(message):
14     ''' Logs the elapsedTime with a given message '''
15     global startTime
16     endTime = datetime.datetime.now()
17     elapsedTime = endTime - startTime
18     hours, remainder = divmod(elapsedTime.seconds, 3600)
19     minutes, seconds = divmod(remainder, 60)
20     totalDays = elapsedTime.days
21     timeStr = str(message) + ': Days: ' + str(totalDays) + \
22             ' hours: ' + str(hours) + ' minutes: ' + str(minutes) + \
23             ' seconds: ' + str(seconds)
24     logStdOut(timeStr)
25     startTime = datetime.datetime.now()

```

```

1 if __name__ == "__main__":
2     logStdOut("LogisticRegression")
3     # 0.0 LinearRegressionTraining

```

```

4   # MeanSquareError learningRate = 0.001, otherwise
5     ↳ overshoots
6
7   # 1 Logistic Regression
8   # 1.1 Binary Classification, only 2 labels
9   # CrossEntropyError, learningRate = 0.01, 98.6% test
10  ↳ accuracy highest
11  # Default Values
12  '''
13
14  numEpoch = 1
15  optimizerType = "adam"
16  executeLinearRegression = False
17  '''

```

```

1  '''
2
3  questionTitle = "1.1.1" # Learning with GD
4  optimizerType = "gd"
5  numEpoch = 201
6  learningRates = [0.1, 0.01, 0.001, 0.0001]
7  weightDecay = 0.01
8  executeLinearRegression = False
9  logStdOut("Starting" + questionTitle)
10 sys.stdout = open("result" + questionTitle + ".txt", "w")
11 ↳ # write a new file from scratch
12 ExecuteBinary(questionTitle, numEpoch, learningRates,
13 ↳ weightDecay, optimizerType, executeLinearRegression)
14 logStdOut("Finished" + questionTitle)
15 #
16 '''

```

```

1   '''
2
3   questionTitle = "1.1.2" # Beyond plain SGD
4   optimizerType = "adam"
5   numEpoch = 200
6   learningRates = [0.1, 0.01, 0.001, 0.0001]
7   weightDecay = 0.01
8   executeLinearRegression = False
9   logStdOut("Starting" + questionTitle)
10  sys.stdout = open("result" + questionTitle + ".txt", "w")
11  ExecuteBinary(questionTitle, numEpoch, learningRates,
12    ↪ weightDecay, optimizerType, executeLinearRegression)
13  logStdOut("Finished" + questionTitle)
14  #

```

```

1   '''
2
3   # 1.1.3
4
5   questionTitle = "1.1.3" # Comparison With Linear
6   ↪ Regression
7   weightDecay = 0.00
8   numEpoch = 200
9   optimizerType = "adam"
10  learningRates = [0.01]
11  executeLinearRegression = True
12  logStdOut("Starting" + questionTitle)
13  sys.stdout = open("result" + questionTitle + ".txt", "w")
14  ExecuteBinary(questionTitle, numEpoch, learningRates,
15    ↪ weightDecay, optimizerType, executeLinearRegression)
16  logStdOut("Finished" + questionTitle)
17  #

```

```

14      # Multi-class Classification, 10 labels
15      # CrossEntropySoftmax Error, learningRate = 0.001
16      questionTitle = "1.2.3" # Logistic Regression for
17          → multiclass
18      weightDecay = 0.01
19      numEpoch = 100 # Multi-class Classification
20      optimizerType = "adam"
21      learningRates = [0.1, 0.01, 0.001]
22      executeLinearRegression = False
23      logStdOut("Starting" + questionTitle)
24      sys.stdout = open("result" + questionTitle + ".txt", "w")
25      ExecuteMulti(questionTitle, numEpoch, learningRates,
26          → weightDecay, optimizerType, executeLinearRegression)
27      logStdOut("Finished" + questionTitle)

28
29 \subsubsection{Part2}

30
31 import datetime
32 import numpy as np
33 import tensorflow as tf
34 import sys
35 import math

```

```

1 class FullyConnectedNeuralNetwork(object):
2     """
3         Fully Connected Neural Network
4         Activation Function: Rectified Linear Unit
5         Cost Function: Cross-Entropy
6         Output Layer: Softmax

```

```

7      Train on entire multi-class MNIST dataset
8      Weight initialization using Xavier Initialization
9      tf.saver to save model at 25%, 50%, 75% and 100% of
10     ↵ training process
11
12  def __init__(self, trainData, trainTarget, validData,
13      ↵ validTarget, testData, testTarget, numEpoch = 1,
14      ↵ learningRate = 0.001, weightDecay = 3e-4, doDropout =
15      ↵ True, doVisualization = False, hiddenLayers =
16      ↵ np.array([1000]), questionTitle=""):
17      """
18          hiddenLayers is an array indicating the number of
19          ↵ hidden units in each layer
20          """
21
22          self.trainData = np.reshape(trainData,
23              ↵ (trainData.shape[0], 784))
24          self.trainTarget = trainTarget
25          self.validData = validData
26          self.validTarget = validTarget
27          self.testData = testData
28          self.testTarget = testTarget
29
30          self.questionTitle = questionTitle
31          self.classifierType = "NN"
32          self.dropoutProbability = 0.5
33          self.doDropout = doDropout
34          self.doVisualization = doVisualization
35
36          self.learningRate = learningRate

```

```

30     self.numEpoch = numEpoch
31
32     self.miniBatchSize = 500
33
34     self.weightDecay = weightDecay
35
36     # Size of array is number of layers
37
38     # values are number of hidden units in each layer
39
40     self.hiddenLayers = hiddenLayers
41
42     # Add the final output as a hidden layer for
43     # multi-class (10 classes)
44
45     self.hiddenLayers = np.append(hiddenLayers, 10)

46
47     self.NeuralNetworkMethod()

```

```

1      def printPlotResults(self, numUpdate, errTrain, accTrain,
2                          errValid, errTest, accValid, accTest, xAxis,
3                          yTrainErr, yValidErr, yTestErr, yTrainAcc, yValidAcc,
4                          yTestAcc, hiddenImages):
5
6      global figureCount
7
8      import matplotlib.pyplot as plt
9
10     print self.classifierType
11
12     removeFinalLayer = self.hiddenLayers[:-1]
13
14     print "HiddenLayer: ", str(removeFinalLayer)
15
16     print "LearningRate: " , self.learningRate, " Mini
17     #batch Size: ", self.miniBatchSize
18
19     print "NumEpoch: ", self.numEpoch
20
21     print "NumUpdate: ", numUpdate
22
23     print "Final Train MSE: ", errTrain
24
25     print "Final Train Acc: ", accTrain
26
27     print "Final Valid MSE: ", errValid

```

```

14     print "Final Test MSE: ", errTest
15     print "Final Valid Acc: ", accValid
16     print "Final Test Acc: ", accTest
17
18     DropoutStr = ""
19
20     if self.doDropout:
21         DropoutStr = "Drop"
22
23     else:
24
25         DropoutStr = ""
26
27     removeFinalLayer = self.hiddenLayers[:-1]
28
29     paramString = "Learn" + str(self.learningRate) +
30
31         ↳ DropoutStr + "Layer" + str(removeFinalLayer) +
32
33         ↳ "Decay" + str(self.weightDecay)
34
35     epochStr = "Epoch" + str(self.numEpoch)
36
37     typeLossStr = "Loss"
38
39     typeAccuracyStr = "Accuracy"
40
41     iterationStr = "Epoch Number"
42
43     trainStr = "Train"
44
45     validStr = "Valid"
46
47     testStr = "Test"
48
49     trainLossStr = trainStr + typeLossStr
50
51     validLossStr = validStr + typeLossStr
52
53     testLossStr = testStr + typeLossStr
54
55
56     trainAccStr = trainStr + typeAccuracyStr
57
58     validAccStr = validStr + typeAccuracyStr
59
60     testAccStr = testStr + typeAccuracyStr
61
62
63     linearLabelStr = "LinearRegression"

```

```
42     figureCount = figureCount + 1
43
44     plt.figure(figureCount)
45
46     title = self.classifierType + typeLossStr + trainStr +
47             → paramString
48
49     plt.title(title)
50
51     plt.xlabel(iterationStr)
52
53     plt.ylabel(typeLossStr)
54
55     plt.plot(np.array(xAxis), np.array(yTrainErr), label =
56             → trainLossStr)
57
58     plt.legend()
59
60     plt.savefig(self.questionTitle + title + epochStr +
61             → ".png")
62
63     plt.close()
64
65     plt.clf()
```

```
53
54     figureCount = figureCount + 1
55
56     plt.figure(figureCount)
57
58     title = self.classifierType + typeLossStr + validStr +
59             → paramString
60
61     plt.title(title)
62
63     plt.xlabel(iterationStr)
64
65     plt.ylabel(typeLossStr)
66
67     plt.plot(np.array(xAxis), np.array(yValidErr), label =
68             → validLossStr)
69
70     plt.legend()
71
72     plt.savefig(self.questionTitle + title + epochStr +
73             → ".png")
74
75     plt.close()
76
77     plt.clf()
```

```

66   figureCount = figureCount + 1
67   plt.figure(figureCount)
68   title = self.classifierType + typeLossStr + testStr +
69     ↪ paramString
70   plt.title(title)
71   plt.xlabel(iterationStr)
72   plt.ylabel(typeLossStr)
73   plt.plot(np.array(xAxis), np.array(yTestErr), label =
74     ↪ testLossStr)
75   plt.legend()
76   plt.savefig(self.questionTitle + title + epochStr +
77     ↪ ".png")
78   plt.close()
79   plt.clf()
80
81
82   figureCount = figureCount + 1
83   plt.figure(figureCount)
84   title = self.classifierType + typeLossStr + trainStr +
85     ↪ validStr + testStr + paramString
86   plt.title(title)
87   plt.xlabel(iterationStr)
88   plt.ylabel(typeLossStr)
89   plt.plot(np.array(xAxis), np.array(yTrainErr), label =
90     ↪ trainLossStr)
91   plt.plot(np.array(xAxis), np.array(yValidErr), label =
92     ↪ validLossStr)
93   plt.plot(np.array(xAxis), np.array(yTestErr), label =
94     ↪ testLossStr)
95   plt.legend()

```

```

89     plt.savefig(self.questionTitle + title + epochStr +
90                 ".png")
91
92     plt.close()
93
94     plt.clf()
95
96
97     # Accuracies
98
99     figureCount = figureCount + 1
100    plt.figure(figureCount)
101    title = self.classifierType + typeAccuracyStr +
102        trainStr + paramStr
103    plt.title(title)
104    plt.xlabel(iterationStr)
105    plt.ylabel(typeAccuracyStr)
106
107    plt.plot(np.array(xAxis), np.array(yTrainAcc), label =
108        trainAccStr)
109
110    accLabel = trainStr + typeAccuracyStr
111    plt.legend()
112
113    plt.savefig(self.questionTitle + title + epochStr +
114                 ".png")
115
116    plt.close()
117
118    plt.clf()
119
120
121    figureCount = figureCount + 1
122    plt.figure(figureCount)
123    title = self.classifierType + typeAccuracyStr +
124        validStr + paramStr
125    plt.title(title)
126    plt.xlabel(iterationStr)
127    plt.ylabel(typeAccuracyStr)
128
129    plt.plot(np.array(xAxis), np.array(yValidAcc), label =
130        validAccStr)

```

```

114     plt.legend()
115
116     plt.savefig(self.questionTitle + title + epochStr +
117                 ".png")
118
119     plt.close()
120
121     plt.clf()
122
123
124     figureCount = figureCount + 1
125
126     title = self.classifierType + typeAccuracyStr +
127                 testStr + paramStr
128
129     plt.figure(figureCount)
130
131     plt.title(title)
132
133     plt.xlabel(iterationStr)
134
135     plt.ylabel(typeAccuracyStr)
136
137     plt.plot(np.array(xAxis), np.array(yTestAcc), label =
138                 testAccStr)
139
140     plt.legend()
141
142     plt.savefig(self.questionTitle + title + epochStr +
143                 ".png")
144
145     plt.close()
146
147     plt.clf()
148
149
150
151     figureCount = figureCount + 1
152
153     title = self.classifierType + typeAccuracyStr +
154                 trainStr + validStr + testStr + paramStr
155
156     plt.figure(figureCount)
157
158     plt.title(title)
159
160     plt.xlabel(iterationStr)
161
162     plt.ylabel(typeAccuracyStr)
163
164     plt.plot(np.array(xAxis), np.array(yTrainAcc), label =
165                 trainAccStr)

```

```

139     plt.plot(np.array(xAxis), np.array(yValidAcc), label =
140             ↳ validAccStr)
141
142     plt.plot(np.array(xAxis), np.array(yTestAcc), label =
143             ↳ testAccStr)
144
145     plt.legend()
146
147     plt.savefig(self.questionTitle + title + epochStr +
148             ↳ ".png")
149
150     plt.close()
151
152     plt.clf()
153
154
155     print self.questionTitle + self.classifierType
156
157     print "Max Test Accuracy is: ",
158             ↳ max(np.array(yTestAcc))
159
160
161     if self.doVisualization:
162
163         figureCount = figureCount + 1
164
165         plt.figure(figureCount)
166
167         # Average across the 1000 hidden units to get a
168             ↳ grayscale image
169
170         # To convert from (784,1000) to (784) to (28,28)
171
172         hiddenImageToPlot =
173
174             ↳ np.reshape(np.average(hiddenImages[0], 1),
175             ↳ (28,28))
176
177         plt.imshow(hiddenImageToPlot,
178             ↳ interpolation="nearest", cmap="gray")
179
180         plt.savefig("FirstHiddenLayerAverageEpoch" +
181             ↳ str(self.numEpoch) + ".png")
182
183         logStdOut('Done Plotting AverageImage')
184
185
186         figureCount = figureCount + 1

```

```

160     plt.figure.figureCount)
161     numHiddenLayer = hiddenImages[0].shape[1]
162     fig = plt.figure(figsize=(28, 28))
163     numCol = 25
164     numRows = numHiddenLayer/numCol
165     for eachHiddenLayer in xrange(numHiddenLayer):
166         logStdOut('hl:' + str(eachHiddenLayer))
167         # Draw each figures (28, 28)
168         currImage = hiddenImages[0]
169         hiddenImageToPlot =
170             ↳ np.reshape(currImage[:, :, eachHiddenLayer:eachHiddenLayer+1],
171             ↳ (28, 28))
172         ax = fig.add_subplot(numCol, numRows,
173             ↳ eachHiddenLayer+1)
174         ax.imshow(hiddenImageToPlot,
175             ↳ interpolation="nearest", cmap="gray")
176         ax.xaxis.set_visible(False)
177         ax.yaxis.set_visible(False)
178         x0, x1 = ax.get_xlim()
179         y0, y1 = ax.get_ylim()
180         ax.set_aspect(abs(x1-x0)/abs(y1-y0))
181         logStdOut('Saving all images')
182         plt.savefig("FirstHiddenLayerAllUnitsEpoch" +
183             ↳ str(self.numEpoch) + ".png")
184         logStdOut('Saved all images')

```

```

1     def ShuffleBatches(self, trainData, trainTarget):
2         # Gets the state as the current time
3         rngState = np.random.get_state()

```

```

4     np.random.shuffle(trainData)
5     np.random.set_state(rngState)
6     np.random.shuffle(trainTarget)
7     return trainData, trainTarget

```

```

1 def layerWiseBuildingBlock(self, inputTensor,
2                             → numberOfHiddenUnits):
3
4     """
5         Layer-wise Building Block
6
7         Input:
8             inputTensor: Hidden activations from previous
9                 → layer
10                numberOfHiddenUnitsInCurrentLayer
11
12        Output: z = Weighted sum of the inputs
13
14        Initialize Weight Matrix and Biases in this function
15
16        A list of hidden activations
17
18        No Loops
19
20        """
21
22        numInput = inputTensor.get_shape().as_list()[1]
23        numOutput = numberOfHiddenUnits
24
25        # Xavier Initialization
26
27        #variance = tf.div(tf.constant(3.0), tf.add(numInput,
28                           → tf.constant(numOutput)))
29
30        variance = 3.0 / (numInput + numOutput)
31
32        weight =
33
34            → tf.Variable(tf.truncated_normal(shape=[numInput,
35                           → numOutput], stddev = math.sqrt(variance)))
36
37        bias = tf.Variable(tf.zeros([numOutput]))
38
39        weightedSum = tf.matmul(tf.cast(inputTensor,
40                           → "float32"), weight) + bias

```

```

20
21     tf.add_to_collection("Weights", weight)
22
23     # Add to collection for weight decay loss
24     weightDecayCoeff =
25         ↳ tf.div(tf.cast(tf.constant(self.weightDecay),
26                         "float32"), tf.constant(2.0))
27
28     # Weight Decay Error calculation
29     weightDecayMeanSquareError =
30         ↳ tf.reduce_mean(tf.square(weight))
31     weightDecayError = tf.multiply(weightDecayCoeff,
32                                     ↳ weightDecayMeanSquareError)
33
34     tf.add_to_collection("WeightDecayLoss",
35         ↳ weightDecayError)
36
37
38     return weightedSum

```

```

1     def buildFullyConnectedNeuralNetwork(self):
2         weightedSum = tf.stack(self.trainData)
3
4         X = tf.placeholder(tf.float32, [None, 784], name='X')
5         y_target = tf.placeholder(tf.float32, [None, 10],
6                                  name='target_y')
7         inputTensor = X
8
9         for currLayer in self.hiddenLayers:
10             weightedSum =
11                 ↳ self.layerWiseBuildingBlock(inputTensor,
12                                             ↳ currLayer)

```

```

10      # Parse with activation function of ReLu
11      inputTensor = tf.nn.relu(weightedSum)
12      # 2.4.1 Dropout
13      if self.doDropout:
14          inputTensor = tf.nn.dropout(inputTensor,
15                                      ↳ self.dropoutProbability)
16
17      # inputTensor is now the final hidden layer, but only
18      ↳ need weighted Sum
19
20      # Need add one more with softmax for output
21      y_predicted = weightedSum
22
23
24
25      # Multi-class Classification
26      correctPred = tf.equal(tf.argmax(y_predicted, 1),
27                             ↳ tf.argmax(y_target, 1))
28      accuracy = tf.reduce_mean(tf.cast(correctPred,
29                                   ↳ "float"))
30
31
32
33      # Cross Entropy Softmax Error Multi-class
34      ↳ Classification
35
36      # note: Cross entropy only works with values from 0 to
37      ↳ 1, so multi-class must be one hot encoded
38
39      crossEntropySoftmaxError =
40
41          ↳ tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
42              logits=weightedSum,
43              ↳ labels=y_target))
44
45      adamOptimizer = tf.train.AdamOptimizer(learning_rate =
46
47          ↳ self.learningRate)
48
49
50      # Calculate weight decay error

```

```

31     totalWeightDecayError =
32         ↳ sum(tf.get_collection("WeightDecayLoss"))
33     crossEntropySoftmaxError =
34         ↳ tf.add(crossEntropySoftmaxError,
35                 ↳ totalWeightDecayError)
36     finalTrainingError = crossEntropySoftmaxError
37
38     train =
39         ↳ adamOptimizer.minimize(loss=finalTrainingError)
40
41
42     weights = tf.get_collection("Weights")
43
44
45
46     return X, y_target, y_predicted, finalTrainingError,
47         ↳ train, accuracy, weights

```

```

1 def NeuralNetworkMethod(self):
2
3     maxTestClassificationAccuracy = 0.0
4
5     inputTensor = tf.stack(self.trainData)
6
7     # Build the fully connected Neural Network
8
9     X, y_target, y_predicted, crossEntropyError, train,
10        ↳ accuracy, weights =
11        ↳ self.buildFullyConnectedNeuralNetwork()
12
13
14     # Session
15
16     init = tf.global_variables_initializer()
17
18     sess = tf.InteractiveSession()
19
20     sess.run(init)
21
22     currEpoch = 0
23
24     xAxis = []
25
26     yTrainErr = []

```

```

14     yValidErr = []
15     yTestErr = []
16     yTrainAcc = []
17     yValidAcc = []
18     yTestAcc = []
19     numUpdate = 0
20     step = 0
21     errTrain = -1
22     errValid = -1
23     errTest = -1
24     accTrain = -1
25     accValid = -1
26     accTest = -1
27     # TODO: 2.4.2 Save model from 25%, 50%, 75% and 100%
28     #       from early stopping point
29     while currEpoch <= self.numEpoch:
30         self.trainData, self.trainTarget =
31             self.ShuffleBatches(self.trainData,
32                                 self.trainTarget)
33         step = 0
34         while step * self.miniBatchSize <
35             self.trainData.shape[0]:
36             # train comes from BuildGraph's optimization
37             #       method
38             # returnedValues =
39             #       sess.run ([whatYouWantToReturnThatWereReturnedFromBuildGraph])
40             #
41             #       feed_dict{valuesToFeedIntoPlaceHoldersThatWereReturnedFromBuildGraph}
42             # sess.run() executes whatever graph you built
43             #       once up to the point where it needs to
44             #       fetch

```

```

36      # and fetches everything that's in
37      ↳ ([variablesToFetch]))
38
39      # Thus, if you don't fetch 'train =
40      ↳ optimizer.minimize(loss)', it won't
41      ↳ optimize it
42
43      _, errTrain, yhat, accTrain, hiddenImages =
44          ↳ sess.run([train, crossEntropyError,
45          ↳ y_predicted, accuracy, weights],
46          ↳ feed_dict={X:
47          ↳ np.reshape(self.trainData[step*self.miniBatchSize:(step+1)*self.
48          ↳ miniBatchSize, 784]), y_target:
49          ↳ self.trainTarget[step*self.miniBatchSize:(step+1)*self.
50          ↳ miniBatchSize, 784]}, step = step + 1
51          numUpdate += 1
52
53          # These will not optimize the function cause
54          ↳ you did not fetch 'train'
55
56          # So it won't have to execute that.
57
58          errValid, accValid =
59              ↳ sess.run([crossEntropyError, accuracy],
60              ↳ feed_dict={X: np.reshape(self.validData,
61              ↳ self.validData.shape[0], 784)), y_target:
62              ↳ self.validTarget})
63
64
65          errTest, accTest =
66              ↳ sess.run([crossEntropyError, accuracy],
67              ↳ feed_dict={X: np.reshape(self.testData,
68              ↳ self.testData.shape[0], 784)), y_target:
69              ↳ self.testTarget})
70
71          # Plot against currEpoch for Neural Network
72
73          xAxis.append(currEpoch)

```

```

48     yTrainErr.append(errTrain)
49     yTrainAcc.append(accTrain)
50     yValidErr.append(errValid)
51     yTestErr.append(errTest)
52     yValidAcc.append(accValid)
53     yTestAcc.append(accTest)
54     currEpoch += 1
55     logStdOut("e" + str(currEpoch))
56
      self.printPlotResults(numUpdate, errTrain, accTrain,
                           ↳ errValid, errTest, accValid, accTest, xAxis,
                           ↳ yTrainErr, yValidErr, yTestErr, yTrainAcc,
                           ↳ yValidAcc, yTestAcc, hiddenImages)

```

```

1 def convertOneHot(targetValues):
2
3     numClasses = np.max(targetValues) + 1
4
5     return np.eye(numClasses)[targetValues]

```

```

1 def executeNeuralNetwork(questionTitle, numEpoch,
2
3     ↳ learningRates, weightDecay, doDropout, doVisualization,
4     ↳ hiddenLayers):
5
6     startTime = datetime.datetime.now()
7
8     with np.load("notMNIST.npz") as data:
9
10        Data, Target = data["images"], data["labels"]
11
12        for learningRate in learningRates:
13
14            for hiddenLayer in hiddenLayers:
15
16                np.random.seed(521)
17
18                randIdx = np.arange(len(Data))
19
20                np.random.shuffle(randIdx)
21
22                Data = Data[randIdx]/255.

```

```

11     Target = Target[randIdx]
12
13     trainData, trainTarget = Data[:15000],
14         ↳ Target[:15000]
15
16     validData, validTarget = Data[15000:16000],
17         ↳ Target[15000:16000]
18
19     # Target values are from 0 to 9
20
21     testData, testTarget = Data[16000:],
22         ↳ Target[16000:]
23
24     trainTarget = convertOneHot(trainTarget)
25
26     validTarget = convertOneHot(validTarget)
27
28     testTarget = convertOneHot(testTarget)
29
30     tf.reset_default_graph()
31
32     FullyConnectedNeuralNetwork(trainData,
33
34         ↳ trainTarget, validData, validTarget,
35
36         ↳ testData, testTarget, numEpoch,
37
38         ↳ learningRate, weightDecay, doDropout,
39
40         ↳ doVisualization, hiddenLayer,
41
42         ↳ questionTitle)
43
44     logElapsedTime(questionTitle +
45
46         ↳ str(learningRate) + str(hiddenLayer))

```

```

1 # Global for logging
2 questionTitle = "" # Need to be global for logging to work
3 startTime = datetime.datetime.now()
4 figureCount = 1
5
6 def logStdOut(message):
7     # Temporary print to std out
8     sys.stdout = sys.__stdout__

```

```

9   print message
10  # Continue editing same file
11  sys.stdout = open("result" + questionTitle + ".txt", "a")
12
13 def logElapsed_time(message):
14     ''' Logs the elapsedTime with a given message '''
15     global startTime
16     endTime = datetime.datetime.now()
17     elapsedTime = endTime - startTime
18     hours, remainder = divmod(elapsedTime.seconds, 3600)
19     minutes, seconds = divmod(remainder, 60)
20     totalDays = elapsedTime.days
21     timeStr = str(message) + ': Days: ' + str(totalDays) + \
22             ' hours: ' + str(hours) + ' minutes: ' + str(minutes) + \
23             ' seconds: ' + str(seconds)
24     logStdOut(timeStr)
25     startTime = datetime.datetime.now()

```

```

1 if __name__ == "__main__":
2     """
3     # 2.2
4     questionTitle = "2.2.2" # No Dropout
5     numEpoch = 30 # Start to early stop at 5 for valid and 8
6     for test
7         learningRates = [0.1, 0.01, 0.001]
8         learningRates = [0.001] # Works better when you run them
9         one by one
10        weightDecay = 3e-4
11        doDropout = False

```

```

10      doVisualization = False
11
12      hiddenLayers = [[1000]]
13
14      logStdOut("Starting" + questionTitle)
15
16      sys.stdout = open("result" + questionTitle + ".txt", "a")
17
18      ↵ # append to file
19
20      executeNeuralNetwork(questionTitle, numEpoch,
21
22      ↵ learningRates, weightDecay, doDropout, doVisualization,
23
24      ↵ hiddenLayers)
25
26      logStdOut("Finished" + questionTitle)
27
28      # '''
29
30      '''
31
32
33      # 2.3.1
34
35      questionTitle = "2.3.1" # Number of hidden units
36
37      numEpoch = 30 # Start to early stop at 5 for valid and 8
38
39      ↵ for test
40
41      learningRates = [0.1, 0.01, 0.001]
42
43      learningRates = [0.001] # Works better when you run them
44
45      ↵ one by one
46
47      weightDecay = 3e-4
48
49      doDropout = False
50
51      doVisualization = False
52
53      hiddenLayers = [[100], [500], [1000]]
54
55      logStdOut("Starting" + questionTitle)
56
57      #sys.stdout = open("result" + questionTitle + ".txt", "w")
58
59      ↵ # write a new file from scratch
60
61      sys.stdout = open("result" + questionTitle + ".txt", "a")
62
63      ↵ # append to file
64
65      executeNeuralNetwork(questionTitle, numEpoch,
66
67      ↵ learningRates, weightDecay, doDropout, doVisualization,
68
69      ↵ hiddenLayers)

```

```

32 logStdOut("Finished" + questionTitle)
33 #
34
35 # 2.3.2
36 questionTitle = "2.3.2" # Number of Layers
37 numEpoch = 30 # Start to early stop at 5 for valid and 8
    ↳ for test
38 learningRates = [0.1, 0.01, 0.001]
39 learningRates = [0.001] # Works better when you run them
    ↳ one by one
40 weightDecay = 3e-4
41 doDropout = False
42 doVisualization = False
43 hiddenLayers = [[500, 500]]
44 logStdOut("Starting" + questionTitle)
45 sys.stdout = open("result" + questionTitle + ".txt", "a")
    ↳ # append to file
46 executeNeuralNetwork(questionTitle, numEpoch,
    ↳ learningRates, weightDecay, doDropout,
    ↳ doVisualization, hiddenLayers)
47 logStdOut("Finished" + questionTitle)
48
49 '''
50 # 2.4 Regularization
51 # Early Stops at about 12 for both valid at test
52 questionTitle = "2.4.1" # Dropout
53 numEpoch = 12
54 learningRates = [0.1, 0.01, 0.001]
55 weightDecay = 3e-4
56 doDropout = True

```

```

57     doVisualization = False
58
59     hiddenLayers = [[1000]]
60
61     logStdOut("Starting" + questionTitle)
62     sys.stdout = open("result" + questionTitle + ".txt", "w")
63     ↵ # write a new file from scratch
64     executeNeuralNetwork(questionTitle, numEpoch,
65     ↵ learningRates, weightDecay, doDropout, doVisualization,
66     ↵ hiddenLayers)
67     logStdOut("Finished" + questionTitle)
68     #
69
70
71     """
72
73     questionTitle = "2.4.2" # Visualization
74     numEpoch = 12
75     learningRates = [0.1]
76     weightDecay = 3e-4
77     doDropout = True
78     doVisualization = True
79     hiddenLayers = [1000]
80
81     logStdOut("Starting" + questionTitle)
82     sys.stdout = open("result" + questionTitle + ".txt", "w")
83     ↵ # write a new file from scratch
84     executeNeuralNetwork(questionTitle, numEpoch,
85     ↵ learningRates, weightDecay, doDropout, doVisualization,
86     ↵ hiddenLayers)
87     logStdOut("Finished" + questionTitle)
88     #
89

```

## 3.2 Entire Code: FuYuan Tee's version

### 3.2.1 Base Code for Q1

```
1 import tensorflow as tf
2
3 import numpy as np
4
5 import math
6
7
8 # Non-interactive plotting
9 import matplotlib.pyplot as plt
10 from IPython import display
11
12 # Interactive plotting
13 from plotly import tools
14 import plotly.plotly as py
15 import plotly.graph_objs as go
16 import plotly.offline as pyo
17 from plotly.offline import download_plotlyjs
18
19 # Configure environment
20 %config InlineBackend.figure_format = 'retina'
21 np.set_printoptions(precision=3)
22
23 # Global Variables
24 CURRENT_DIR =
    ↵ '/Users/christophertee/Dropbox/University/MASc/Courses/Winter
    ↵ 2017' + \
```

```

25  '/ECE521 (Inference Algorithms & Machine Learning) /Assignment
26    ↵ 2'
27
28 # Activate Plotly Offline for Jupyter
29 pyo.init_notebook_mode(connected=True)
30
31 # Load notMNIST dataset
32 '''
33 Training Set: 3500 images
34 Validation Set: 100 images
35 Test Set: 145 images
36 Images are 28 x 28 (normalised) pixels
37 '''
38 with np.load("notMNIST.npz") as data:
39     Data, Target = data["images"], data["labels"]
40
41     # Subsetting data for classes 'C' (2) and 'J' (9)
42     posClass = 2 # 'C'
43     negClass = 9 # 'J'
44     dataIndx = (Target==posClass) + (Target==negClass)
45
46     Data = Data[dataIndx] / 255.
47     Target = Target[dataIndx].reshape(-1, 1)
48
49     # Converts target labels to 'C' (0) and 'J' (1)
50     Target[Target==posClass] = 1
51     Target[Target==negClass] = 0
52
53     # Set random seed

```

```

54 np.random.seed(521)

55

56 # Generate and shuffle random index
57 randIdx = np.arange(len(Data))
58 np.random.shuffle(randIdx)

59

60 Data = Data[randIdx]
61 Target = Target[randIdx]

62

63 # Flatten arrays of dimension m x 28 x 28 into array of
64 # dimension m x 784
65 Data = Data.reshape(Data.shape[0], -1)

66

67 # Standardizing inputs of dataset
68 Data -= np.mean(Data, axis=0)
69 Data /= np.std(Data, axis=0)

70

71 # Partition data into training, validation and test
72 # datasets
73 Data, Target = Data[randIdx], Target[randIdx]
74 trainData, trainTarget = Data[:3500], Target[:3500]
75 validData, validTarget = Data[3500:3600],
76 # Target[3500:3600]
77 testData, testTarget = Data[3600:], Target[3600:]

78 MAX_ITER = 10000

79 def tuneLearningRate(etaList, model, section_dir,
80 # batchSize=500, lambda_=0.01, opt='Adam'):
81 # Returns the i-th batch of training data and targets
82 # Generates a new, reshuffled batch once all previous
83 # batches are fed

```

```

80  def getNextTrainingBatch (currentIter, randIdx):
81      currentBatchNum = currentIter % (trainData.shape[0] /
82          ↳ batchSize)
83
84      if currentBatchNum == 0:
85          np.random.shuffle(randIdx)
86
87          lowerBoundIdx = currentBatchNum * batchSize
88          upperBoundIdx = (currentBatchNum + 1) * batchSize
89
90          return
91              ↳ trainData[randIdx[lowerBoundIdx:upperBoundIdx]],
92              ↳ trainTarget[randIdx[lowerBoundIdx:upperBoundIdx]]
93
94
95
96      # Generate updated plots for training and validation MSE
97      def plotErrGraph (errList, param):
98          label = '$\eta$ = ' + str(param)
99          label_classification = ['train.', 'valid.']
100
101
102          display.clear_output(wait=True)
103          plt.figure(figsize=(8,5), dpi=200)
104
105
106          for i, err in enumerate(errList):
107              plt.plot(range(len(err)), err, '-',
108                      label=label+' '+label_classification[i])
109
110
111          plt.axis([0, MAX_ITER, 0, np.amax(errList)])
112          plt.legend()
113          plt.show()
114
115
116          # Calculates the ratio between the n-th average epoch MSE
117          ↳ and the (n-1)-th average epoch MSE
118
119          def ratioAverageEpochMSE (currentValidErr):

```

```

105     averageN =
106         ↳ np.average(currentValidErr[-(np.arange(epochSize -
107             ↳ 1) + 1)])
108
109     averageNlessOne =
110         ↳ np.average(currentValidErr[-(np.arange(epochSize -
111             ↳ 1) + epochSize)])
112
113     return averageN / averageNlessOne
114
115
116
117
118     # Returns True if the average epoch validation MSE is at
119     ↳ least 99.9% of the previous epoch average.
120
121     # i.e. Returns True if the average learnings between epoch
122     ↳ is less than +0.1%
123
124     # Otherwise, returns False
125
126     def shouldStopEarly(currentValidErr):
127
128         if currentValidErr.shape[0] < 2 * epochSize:
129
130             return False
131
132         return True if (ratioAverageEpochMSE(currentValidErr)
133             ↳ > 0.999) else False
134
135
136
137
138     # Start of function
139
140     summaryList = []
141
142     randIdx = np.arange(trainData.shape[0])
143
144     epochSize = trainData.shape[0] / batchSize
145
146     randIdx = np.arange(trainData.shape[0])
147
148
149     assert section_dir
150
151     chapter_dir = '/Binary Loss/Section ' + section_dir + '/'
152
153     currentTime = '{:%b%d'
154
155         ↳ %H_%M_%S}'.format(datetime.datetime.now())

```

```

127
128     for eta in etaList:
129         # Reset graph to prevent duplication of ops and
130         # variables
131         tf.reset_default_graph()
132
133         # Build new graph
134         assert model in ['linear', 'logistic', 'softmax']
135
136         if model == 'linear':
137             W, b, X, Y, YhatThres, error, accuracy, optimizer
138             = buildLinearGraph(eta, lambda_)
139
140         elif model == 'logistic':
141             W, b, X, Y, YhatThres, error, accuracy, optimizer
142             = buildLogisticGraph(eta, lambda_, opt)
143
144         elif model == 'softmax':
145             W, b, X, Y, YhatThres, error, accuracy, optimizer
146             = buildSoftmaxGraph(eta, lambda_)
147
148         # Begin session
149         with
150             tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
151             log_device_placement=False)) as sess:
152             # Log starting time
153             startTime = time.time()
154
155
156             # Create summary writer
157             writer = tf.summary.FileWriter(CURRENT_DIR +
158                 LOG_DIR + chapter_dir + current_time + \
159                 '/Eta-' + str(eta),
160                 )

```

```

149                                         graph=sess.graph)

150

151     # Initialise all TensorFlow variables
152     tf.global_variables_initializer().run()

153

154     # Creates blank training and validation MSE arrays
155     # for the Session
156
157     currentTrainErr = np.array([])[:, np.newaxis]
158
159     currentValidErr = np.array([])[:, np.newaxis]
160
161     currentTestErr = np.array([])[:, np.newaxis]

162

163     # Runs update
164
165     currentIter = 0
166
167     while currentIter < MAX_ITER:
168
169         inputData, inputTarget =
170             getNextTrainingBatch(currentIter, randIdx)
171
172
173         _, trainErr, trainAcc = sess.run([optimizer,
174             error, accuracy], feed_dict={X: inputData,
175             Y: inputTarget})
176
177         validErr, validAcc = sess.run([error,
178             accuracy], feed_dict={X: validData, Y:
179             validTarget})
180
181         testErr, testAcc = sess.run([error, accuracy],
182             feed_dict={X: testData, Y: testTarget})

```

```

172         currentTrainErr = np.append(currentTrainErr,
173             ↵ trainErr)
174         currentValidErr = np.append(currentValidErr,
175             ↵ validErr)
176         currentTestErr = np.append(currentTestErr,
177             ↵ testErr)
178
179
180         # Update graph of training and validation MSE
181         ↵ arrays
182     #             if (currentIter < 3) or (currentIter % 1000
183         ↵ == 0):
184
185     #             plotErrGraph([currentTrainErr,
186         ↵ currentValidErr], eta)
187
188     # At every epoch, check for early stopping
189     ↵ possiblty. If so, breaks from while loop
190     if currentIter % epochSize == 0:
191
192         if shouldStopEarly(currentValidErr):
193
194             writer.close()
195
196             break
197
198
199         currentIter += 1

```

```

192         if currentIter == MAX_ITER:
193             writer.close()
194
195     # Save session results as dictionary and appends to
196     # → MSEsummaryList
197     summaryList.append(
198         {
199             'eta': eta,
200             'B': batchSize,
201             'lambda': lambda_,
202             'optimizer': opt,
203             'numIter': currentIter + 1,
204             'epoch': float(currentIter + 1) / epochSize,
205             'trainError': currentTrainErr,
206             'validError': currentValidErr,
207             'testError': currentTestErr,
208             'trainAccuracy': currentTrainAcc,
209             'validAccuracy': currentValidAcc,
210             'testAccuracy': currentTestAcc
211         }
212     )
213
214     # Print stats when one eta run is done
215     print 'eta: %7.3f, numIter: %7d, validError: %.3f,'
216     # → testAcc: %.3f duration: %3.1fs' % \
217     #   (summaryList[-1]['eta'],
218     #    → summaryList[-1]['numIter'],
219     #    → summaryList[-1]['validError'][-1], \
220     #    → np.mean(summaryList[-1]['testAccuracy'][-epochSize:])),
221     #    → time.time() - startTime)

```

```
217  
218     return summaryList
```

### 3.2.2 Q1.1.1: Learning

```
1  # TensorFlow logistic model  
2  def buildLogisticGraph(eta, lambda_, opt):  
3      assert opt in ['GD', 'Adam']  
4      with tf.device('/cpu:0'):  
5          with tf.variable_scope('logistic_regression'):  
6              # Model inputs  
7              with tf.name_scope('placeholders'):  
8                  X = tf.placeholder(tf.float32, shape=[None,  
9                                     ↵ None], name='input')  
10                 Y = tf.placeholder(tf.float32, shape=[None,  
11                                     ↵ None], name='target')  
12  
13                 # Model parameters  
14                 with tf.name_scope('parameters'):  
15                     W = tf.get_variable('weights', shape=[784, 1],  
16                                     ↵ initializer=tf.truncated_normal_initializer(stddev=0.  
17                     b = tf.get_variable('biases', shape=[1, 1],  
18                                     ↵ initializer=tf.constant_initializer(0.0))  
19  
20                     with tf.device('/cpu:0'):  
21                         # Predicted target  
22                         with tf.name_scope('prediction'):  
23                             Z = tf.add(tf.matmul(X, W), b, 'logits')  
24                             Yhat = tf.sigmoid(Z, name='pred')
```

```

21
22     # Metrics
23
24     with tf.name_scope('metrics'):
25
26         with tf.name_scope('error'):
27
28             error =
29
30                 ↳ tf.add(tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
31                     ↳ labels=Y, name='elem_x_entropy'),
32
33                     ↳ \name=name='total_x_entropy',
34
35                     ↳ \
36
37                     ↳ tf.multiply(lambda_ /
38                         ↳ 2,
39
40                         ↳ tf.matmul(tf.transpose(W),
41
42                             ↳ W),
43
44                             ↳ name='l2_loss'), \
45
46                             ↳ name=name='total_loss')
47
48             with tf.name_scope('threshold'):
49
50                 YhatThres =
51
52                     ↳ tf.cast(tf.greater_equal(Yhat,
53                         ↳ 0.5, name='pred_thres'),
54
55                         ↳ tf.float32)
56
57             with tf.name_scope('accuracy'):
58
59                 accuracy =
60
61                     ↳ tf.truediv(tf.reduce_sum(tf.cast(tf.equal(YhatThres,
62
63                         ↳ Y), tf.int32),
64
65                         ↳ name='total_matches'), \
66
67                         ↳ tf.shape(X)[0],
68
69                         ↳ \
70
71                         ↳ name=name='accuracy')

```

```

35
36     # Optimizer
37     with tf.device('/cpu:0'):
38         if opt == 'GD':
39             optimizer =
40                 → tf.train.GradientDescentOptimizer(eta).minimize(error)
41         elif opt == 'Adam':
42             optimizer =
43                 → tf.train.AdamOptimizer(eta).minimize(error)
44
45     return W, b, X, Y, YhatThres, error, accuracy, optimizer
46
47 etaList = [0.001, 0.01, 0.1, 1, 10]
48 summary1_1 = tuneLearningRate(etaList, model='logistic',
49     → section_dir='1.1', opt='GD')

```

### 3.2.3 Q.1.1.2: Beyond Plain SGD

```

1 etaList = [1e-4, 1e-3, 1e-2, 1e-1, 1]
2 summary1_2 = tuneLearningRate(etaList, model='logistic',
3     → section_dir='1.2', opt='Adam')

```

### 3.2.4 Q1.1.3 Comparison with linear regression

```

1 # TensorFlow Linear Graph
2 def buildLinearGraph(eta, lambda_):
3     with tf.device('/cpu:0'):

```

```
4   with tf.variable_scope('linear_regression'):  
5       # Model inputs  
6  
7       with tf.name_scope('placeholders'):  
8           X = tf.placeholder(tf.float32, shape=[None,  
9                           ↳ None], name='Input')  
10          Y = tf.placeholder(tf.float32, shape=[None,  
11                           ↳ None], name='Target')  
12  
13  
14      # Model parameters  
15      with tf.name_scope('parameters'):  
16          W = tf.get_variable('weights', shape=[784, 1],  
17                           ↳ initializer=tf.truncated_normal_initializer(stddev=0.  
18          b = tf.get_variable('biases', shape=[1, 1],  
19                           ↳ initializer=tf.constant_initializer(0.0))  
20  
21      with tf.device('/cpu:0'):  
22          # Predicted target  
23          with tf.name_scope('prediction'):  
24              Yhat = tf.add(tf.matmul(X, W), b, 'pred')  
25  
26          # Metrics  
27          with tf.name_scope('metrics'):  
28              with tf.name_scope('error'):  
29                  error =  
30                      ↳ tf.add(tf.reduce_mean(tf.nn.l2_loss(tf.subtract  
31                      ↳ Y), name='elem_l2_loss'), \  
32                          ↳ name='total_l2_'  
33  
34  
35          tf.multiply(lambda_ /  
36                          ↳ 2,  
37                          ↳ tf.matmul(tf.transpose(W),  
38                          ↳ W),  
39                          ↳ name='l2_loss'), \  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
999  
1000
```

```

26                                         name='total_loss')

27     with tf.name_scope('threshold'):

28         YhatThres =
29             ↳ tf.cast(tf.greater_equal(Yhat,
30                                         ↳ 0.5, name='pred_thres'),
31                                         ↳ tf.float32)
32
33         with tf.name_scope('accuracy'):
34
35             accuracy =
36                 ↳ tf.truediv(tf.reduce_sum(tf.cast(tf.equal(Yhat,
37                                         ↳ Y), tf.int32),
38                                         ↳ name='total_matches'), \
39                                         tf.shape(X)[0],
40                                         ↳ \
41                                         name='accuracy')
42
43
44     # Optimizer
45
46     with tf.device('/cpu:0'):

47         optimizer =
48             ↳ tf.train.AdamOptimizer(eta).minimize(error)
49
50
51     return W, b, X, Y, YhatThres, error, accuracy, optimizer
52
53
54 etaList = [1e-4, 1e-3, 1e-2, 1e-1, 1, 10]
55
56 print 'Linear Regression:'
57
58 summary1_3_linear = tuneLearningRate(etaList, model='linear',
59                                     ↳ section_dir='1.3Lin', lambda_=0.0)
60
61 print '\nLogistic Regression:'
62
63 summary1_3_logistic = tuneLearningRate(etaList,
64                                         ↳ model='logistic', section_dir='1.3Log', lambda_=0.0)

```

```

46 def generateDummyLosses():
47     # Generate dummy data
48     Yhat = np.linspace(0.01, 1, 100)[:, np.newaxis]
49     Ydummy = 0.0
50
51     # Calculate losses
52     xEntropy = - Ydummy * np.log(Yhat) - (1 - Ydummy) *
53         np.log(1 - Yhat)
54     squaredErr = (Yhat - Ydummy)**2
55
56     # Generate plot traces
57     xEntropyTrace = go.Scatter(
58         x = Yhat,
59         y = xEntropy,
60         name = 'Cross-Entropy'
61     )
62     squareErrTrace = go.Scatter(
63         x = Yhat,
64         y = squaredErr,
65         name = 'Squared-Error'
66     )
67
68     data = go.Data([xEntropyTrace, squareErrTrace])
69
70     # Generate figure layout
71     layout = go.Layout(
72         title = '$\\text{Comparison of Cross Entropy and}$',
73         '$\\text{Squared Error Losses on Dummy Variable } y = 0$',',
74         xaxis = { 'title': '$\\hat{y}$' },
75         yaxis = { 'title': 'Loss' }

```

```

74 )
75
76     # Creates and plots figure
77     figure = go.Figure(data=data, layout=layout)
78     return py.iplot(figure,
79                     ↵ filename='A2Q1.1.3_LossComparison')
80
81 fig1_3 = generateDummyLosses()
82 fig1_3

```

### 3.2.5 Q1.2.3: Multi-class classification

```

1  # TensorFlow Softmax Model
2  def buildSoftmaxGraph(eta, lambda_) :
3      with tf.device('/cpu:0') :
4          with tf.variable_scope('softmax') :
5              # Model inputs
6              with tf.name_scope('placeholders') :
7                  X = tf.placeholder(tf.float32, shape=[None,
8                                  ↵ None], name='input')
9                  Y = tf.placeholder(tf.float32, shape=[None,
10                                 ↵ None], name='target')
11
12              # Model parameters
13              with tf.name_scope('parameters') :
14                  W = tf.get_variable('weights', shape=[784,
15                                  ↵ 10],
16                                  ↵ initializer=tf.truncated_normal_initializer(stddev=0.1))
17                  b = tf.get_variable('biases', shape=[1, 10],
18                                  ↵ initializer=tf.constant_initializer(0.0))

```

```

13
14   with tf.device('/cpu:0'):
15     # Predicted target
16     with tf.name_scope('prediction'):
17       Z = tf.add(tf.matmul(X, W), b, 'logits')
18       Yhat = tf.nn.softmax(Z, name='activation')
19
20     # Metrics
21     with tf.name_scope('metrics'):
22       with tf.name_scope('error'):
23         with tf.name_scope('x_entropy'):
24           x_entropy =
25             ↳ tf.reduce_mean(tf.nn.softmax_cross_entropy_with_
26
27             ↳ \_
28
29             ↳ name='total_x_entropy'
30
31       with tf.name_scope('l2_loss'):
32         l2_loss = tf.multiply(lambda_ / 2, \
33
34           ↳ tf.reduce_sum(tf.matmul(tf.
35             ↳ W)), \
36             ↳ name='l2_loss')
37
38       with tf.name_scope('total_loss'):
39         error = tf.add(x_entropy, l2_loss,
40             ↳ name='total_loss')

```

```

33     with tf.name_scope('threshold'):
34         YhatThres = tf.cast(tf.argmax(Yhat, axis=1,
35                               name='pred_threshold'), tf.float32)
36     with tf.name_scope('accuracy'):
37         Ycollapsed = tf.cast(tf.argmax(Y, axis=1),
38                               tf.float32, name='target_reverse_one_hot')
39         accuracy =
40             tf.truediv(tf.reduce_sum(tf.cast(tf.equal(YhatThres,
41                                             Ycollapsed) \
42                                             , tf.int32),
43                                         name='total_match') \
44                                         , tf.shape(X)[0], \
45                                         name='accuracy')
46
47     return W, b, X, Y, YhatThres, error, accuracy, optimizer
48
49 # Load notMNIST dataset
50 '''
51 Training Set: 15,000 images
52 Validation Set: 1,000 images
53 Test Set: 2,724 images

```

```

54     Images are 28 x 28 (normalised) pixels
55
56 with np.load("notMNIST.npz") as data:
57
58     Data, Target = data["images"], data["labels"]
59
60     # Set random seed
61     np.random.seed(521)
62
63     # Generate and shuffle random index
64     randIdx = np.arange(len(Data))
65     np.random.shuffle(randIdx)
66
67     Data = Data[randIdx]/255.
68
69     Target = Target[randIdx]
70
71     # Generates one-hot version of target
72     oneHot = np.zeros((Target.shape[0], 10))
73     oneHot[np.arange(Target.shape[0]), Target] = 1
74     Target = oneHot
75
76
77     # Flatten arrays of dimension m x 28 x 28 into array of
78         → dimension m x 784
79     Data = Data.reshape(Data.shape[0], -1)
80
81     # Standardizing inputs of dataset
82     Data -= np.mean(Data, axis=0)
83     Data /= np.std(Data, axis=0)
84
85     # Partition data into training, validation and test
86         → datasets

```

```

82     trainData, trainTarget = Data[:15000], Target[:15000]
83
84     validData, validTarget = Data[15000:16000],
85         ↳ Target[15000:16000]
86
87     testData, testTarget = Data[16000:], Target[16000:]

```

### 3.2.6 Base Code for Q2

```

1 import tensorflow as tf
2
3 import numpy as np
4
5 import math
6
7
8 # Non-interactive plotting
9 import matplotlib.pyplot as plt
10 from IPython import display
11
12 # Interactive plotting
13 from plotly import tools
14 import plotly.plotly as py
15 import plotly.graph_objs as go
16 import plotly.offline as pyo
17 from plotly.offline import download_plotlyjs
18
19 # Configure environment
20 %config InlineBackend.figure_format = 'retina'
21 np.set_printoptions(precision=3)
22

```

```

23 # Global Variables
24 CURRENT_DIR =
25     ↳ '/Users/christophertee/Dropbox/University/MASc/Courses/Winter
26     ↳ 2017' + \
27         '/ECE521 (Inference Algorithms & Machine Learning)/Assignment
28         ↳ 2'
29 LOG_DIR = '/Logs'
30
31 # Activate Plotly Offline for Jupyter
32 pyo.init_notebook_mode(connected=True)
33
34 # Load notMNIST dataset
35 """
36 Training Set: 15,000 images
37 Validation Set: 1,000 images
38 Test Set: 2,724 images
39 Images are 28 x 28 (normalised) pixels
40 """
41
42 with np.load("notMNIST.npz") as data:
43     Data, Target = data["images"], data["labels"]
44
45     # Set random seed
46     np.random.seed(521)
47
48     # Generate and shuffle random index
49     randIdx = np.arange(len(Data))
50     np.random.shuffle(randIdx)

51
52     Data = Data[randIdx]/255.
53     Target = Target[randIdx]

```

```

50
51     # Generates one-hot version of target
52     Target = np.eye(10)[Target]
53     oneHot = np.zeros((Target.shape[0], 10))
54     oneHot[np.arange(Target.shape[0]), Target] = 1
55     Target = oneHot
56
57     # Flatten arrays of dimension m x 28 x 28 into array of
58     # dimension m x 784
59     Data = np.reshape(Data, (Data.shape[0], 784))
60
61     # Standardizing inputs of dataset
62     Data -= np.mean(Data, axis=0)
63     Data /= np.std(Data, axis=0)
64
65     # Partition data into training, validation and test
66     # datasets
67     trainData, trainTarget = Data[:15000], Target[:15000]
68     validData, validTarget = Data[15000:16000],
69     # Target[15000:16000]
70     testData, testTarget = Data[16000:], Target[16000:]
71
72     # TensorFlow Fully-Connected Neural Network
73     def build_FCNN(eta, lambda_, numHidLayers, layerUnits,
74     # dropout_rate):
75         NUM_FEATURES = 784
76         NUM_TARGET_CATEGORIES = 10
77         '''
78             - Builds a fully-connected neural network with
79             numHidLayers hidden layers, with

```

```

75     layerUnits[i] in the i-th hidden layer

76

77     Inputs:
78         eta: Learning rate
79         lambda_: L2 regularizer
80         numHidLayers: Number of hidden layers
81         layerUnits: List of number of units in each layer;
82         ↵ len(layerUnits) == numHidLayer
83         '''
84
85
86     def connect_layers(h, numNextUnit, name='input_to_layer'):
87         '''
88             - Creates weights and biases to connect previous layer
89             to next layer
90             - Returns the weighted sum of inputs for to the next
91             layer
92             - Initializes weights using Xavier initialization.
93
94     Inputs:
95         hIn: Incoming activation tensor from previous
96         layer
97         numOut: Number of hidden units of next layer
98
99     Outputs:
100        z: Weighted sum of inputs into the next layer
101        '''
102
103        numPrevUnit = h.get_shape()[1]
104
105        # Xavier initialization for weights
106        xavier = tf.sqrt(3 / tf.cast(numPrevUnit +
107            ↵ numNextUnit, tf.float32), name='xavier')

```

```

101
102     W = tf.get_variable('weights', shape=[numPrevUnit,
103                           ↳ numNextUnit], \
104
105                           ↳ initializer=tf.truncated_normal_initializer(
106                           ↳ seed=1))
107
108     b = tf.get_variable('biases', shape=[1, numNextUnit],
109                           ↳ initializer=tf.constant_initializer(0.0))
110
111
112     tf.add_to_collection(tf.GraphKeys.WEIGHTS, W)
113     tf.add_to_collection(tf.GraphKeys.BIASES, b)
114
115
116     return tf.add(tf.matmul(h, W), b, name=name)
117
118
119
120
121
122
123
124
125

```

**def build\_layers**(numHidLayers, layerUnits, dropout\_rate):  
 '''  
 - Builds hidden layers and output layer  
 '''  
**Inputs:**  
 numHidLayers: Number of hidden layers  
 layerUnits: List of number of units in each layer;  
 ↳ len(layerUnits) == numHidLayer  
**Output:**  
 zOut: Logits to output layer  
 '''  
 ### Builds hidden layers  
 ZList = [] # Stores the inputs to the i-th layer  
 hList = [] # Stores the activation of the i-th layer

```

126
127     for i in range(numHidLayers):
128         # Create the i-th hidden layer
129         with tf.variable_scope('hidden_layer_%d' % i) as
130             → scope:
131                 # Calculates input to the i-th hidden layer
132                 # Uses placeholder tensor X instead of Z_(i-1)
133                     → in connectLayers for the first hidden
134                     → layer
135
136         if i == 0:
137             ZList.append(connect_layers(X,
138                             → layerUnits[i]]))
139
140         else:
141             ZList.append(connect_layers(hList[-1],
142                             → layerUnits[i]]))
143
144
145             # Calculates ReLU activation of each layer
146
147             → hList.append(tf.nn.dropout(tf.nn.relu(ZList[i]),
148             → name='hidden_activation'),
149             → keep_prob=1-dropout_rate, \
150             → name='dropped_activations'))
151
152
153     # Adds activation variables into GraphKey.ACTIVATIONS
154     → collection
155
156     for h in hList:
157
158         tf.add_to_collection(tf.GraphKeys.ACTIVATIONS, h)
159
160
161     ### Builds output layer

```

```

146     with tf.variable_scope('output_layer'):

147         ZList.append(connect_layers(hList[-1],
148                           NUM_TARGET_CATEGORIES, name='logits'))

149
150
151
152 ######
153 #           Function begins          #
154 ######
155 mode = '/gpu:0'
156 tf.set_random_seed(521)
157
158 assert len(layerUnits) == numHidLayers
159
160 with tf.device('/cpu:0'):

161     with tf.name_scope('placeholders'):

162         X = tf.placeholder(tf.float32, shape=[None,
163                           NUM_FEATURES], name='input_layer')
164
165         Y = tf.placeholder(tf.float32, shape=[None,
166                           NUM_TARGET_CATEGORIES], name='target')
167
168     with tf.device(mode):
169
170         logits = build_layers(numHidLayers, layerUnits,
171                           dropout_rate) # Logits to output layer
172
173
174     # Calculates prediction
175
176     with tf.name_scope('prediction'):

177         Yhat = tf.nn.softmax(logits, name='pred')
178
179         YhatThres = tf.cast(tf.argmax(Yhat, axis=1,
180                           name='pred_threshold'), tf.float32)

```

```

172
173     with tf.device('/cpu:0'):
174         # Calculates training metrics
175         with tf.name_scope('metrics'):
176             # Calculates loss function
177             with tf.name_scope('loss'):
178                 # Calculates cross-entropy
179                 with tf.name_scope('xent'):
180                     xent =
181                         ↳ tf.reduce_mean(tf.nn.softmax_cross_entropy_with_l
182                                         ↳ \
183                                         ↳ name='total_x_entropy')
184             # Calculates l2 loss
185             with tf.name_scope('l2_loss'):
186                 weights =
187                     ↳ tf.get_collection_ref(tf.GraphKeys.WEIGHTS)
188                 biases =
189                     ↳ tf.get_collection_ref(tf.GraphKeys.BIASES)
190                 l2_loss = tf.multiply(lambda_ / 2, \
191                                         ↳ tf.add_n([tf.nn.l2_loss(W
192                                         ↳ for W in
193                                         ↳ weights)],
194                                         ↳ name='total_l2_loss')

```



```

210     MAX_ITER = 3000
211
212     """
213     - Returns the i-th batch of training data and targets
214     - Generates a new, reshuffled batch once all previous
215     ↳ batches are fed
216     """
217
218     def get_next_training_batch(currentIter, randIdx):
219         currentBatchNum = currentIter % (trainData.shape[0] /
220             ↳ batchSize)
221
222         if currentBatchNum == 0:
223             np.random.shuffle(randIdx)
224
225         lowerBoundIdx = currentBatchNum * batchSize
226
227         upperBoundIdx = (currentBatchNum + 1) * batchSize
228
229         return
230             ↳ trainData[randIdx[lowerBoundIdx:upperBoundIdx]],
231             ↳ trainTarget[randIdx[lowerBoundIdx:upperBoundIdx]]
232
233
234     # Generate updated plots for training and validation MSE
235
236     def plot_err_graph(errList, param):
237
238         label = '$\eta$ = ' + str(param)
239
240         label_classification = ['train.', 'valid.']
241
242
243         display.clear_output(wait=True)
244
245         plt.figure(figsize=(8,5), dpi=200)
246
247
248         for i, err in enumerate(errList):
249
250             plt.plot(range(len(err)), err, '-',
251             ↳ label=label+' '+label_classification[i])
252
253
254         plt.axis([0, MAX_ITER, 0, np.amax(errList)])

```

```

235     plt.legend()
236     plt.show()
237
238     # Calculates the ratio between the n-th average epoch MSE
239     # and the (n-1)-th average epoch MSE
240
241     def average_epoch_err_ratio(validErr):
242
243         averageN = np.mean(validErr[-epochSize:])
244
245         averageNlessOne =
246             np.mean(validErr[-epochSize*2:-epochSize])
247
248         return averageN / averageNlessOne
249
250
251     # Returns True if the average epoch validation MSE is at
252     # least 99.9% of the previous epoch average.
253
254     # i.e. Returns True if the average learnings between epoch
255     # is less than +0.1%
256
257     # Otherwise, returns False
258
259     def should_stop_early(validErr):
260
261         if validErr.shape[0] < 2 * epochSize:
262
263             return False
264
265         return True if (average_epoch_err_ratio(validErr) > 1)
266             else False
267
268
269     # Start of function
270
271     summaryList = []
272
273     epochSize = trainData.shape[0] / batchSize
274
275
276     assert sectionDir
277
278
279     for eta in etaList:
280
281         for i, layerUnits in enumerate(layerUnitsList):

```

```

260     numHidLayers = numHidLayersList[i]
261
262     # Set random seed
263     np.random.seed(521)
264     randIdx = np.arange(trainData.shape[0])
265
266     # Prints run hyperparameters
267     print 'Eta: ', eta
268     print 'Hidden Layers: ', numHidLayers
269     print 'Units in Layers: ', layerUnits
270     print 'Dropout Rate: ', dropout_rate
271
272     # Create directory paths
273     chapterDir = '/Neural Network/Section ' +
274         ↳ sectionDir + '/Eta_' + str(eta) + '/'
275     currentTime = '{:b%d' +
276         ↳ '%H_%M_%S}'.format(datetime.datetime.now())
277     writeDir = CURRENT_DIR + LOG_DIR + chapterDir +
278         ↳ currentTime
279
280     # Build new graph
281     tf.reset_default_graph() # Reset graph to prevent
282         ↳ duplication of ops and variables
283     X, Y, YhatThres, W, error, accuracy, optimizer =
284         ↳ build_FCNN(eta, lambda_, numHidLayers,
285         ↳ layerUnits, dropout_rate)
286
287     # Create saver
288     #     saver = tf.train.Saver()
289
290     # Begin session

```

```

284
285     with tf.Session(config=tf.ConfigProto(log_device_placement=False)) as sess:
286
287         # Log starting time
288         startTime = time.time()
289
290
291         # Create summary writer
292         writer = tf.summary.FileWriter(writeDir,
293                                         graph=sess.graph)
294
295         # Initialise all TensorFlow variables
296         tf.global_variables_initializer().run()
297
298
299         # Create blank training, validation and test
300         # error arrays for the Session
301         trainErr = np.array([])[:, np.newaxis]
302         validErr = np.array([])[:, np.newaxis]
303         testErr = np.array([])[:, np.newaxis]
304
305         # Create blank training, validation and test
306         # accuracies arrays for the Session
307         trainAcc = np.array([])[:, np.newaxis]
308         validAcc = np.array([])[:, np.newaxis]
309         testAcc = np.array([])[:, np.newaxis]
310
311
312         # Create blank arrays to store weights for the
313         # Session
314         weights_10 = np.array([])[:, np.newaxis]
315
316
317         # Runs update

```

```

308     currentIter = 0
309
while currentIter < MAX_ITER:
310
    # To be memory efficient, save weights
    # → only at the start of each epoch
311
    if currentIter % epochSize == 0:
312
        if currentIter == 0:
313
            weights_10 = sess.run(W) [0] [:,:,,
            # → np.newaxis]
314
        else:
315
            weights_10 = np.append(weights_10,
            # → sess.run(W) [0] [:,:,,
            # → np.newaxis], axis=2)
316
317
        # Create training mini-batch
        # → stochastically from data
318
        inputData, inputTarget =
        # → get_next_training_batch(currentIter,
        # → randIdx)
319
320
        # Calculate error and accuracy metrics
        # → before optimization routine
321
        currTrainErr, currTrainAcc =
        # → sess.run([error, accuracy]),
        # → feed_dict={X: inputData, Y:
        # → inputTarget})
322
        currValidErr, currValidAcc =
        # → sess.run([error, accuracy]),
        # → feed_dict={X: validData, Y:
        # → validTarget})
323
        currTestErr, currTestAcc =
        # → sess.run([error, accuracy]),
        # → feed_dict={X: testData, Y:
        # → testTarget})

```

```

324
325         # Append run values
326         trainErr = np.append(trainErr,
327                               ↳ currTrainErr)
328         validErr = np.append(validErr,
329                               ↳ currValidErr)
330         testErr = np.append(testErr, currTestErr)

331
332         trainAcc = np.append(trainAcc,
333                               ↳ currTrainAcc)
334         validAcc = np.append(validAcc,
335                               ↳ currValidAcc)
336         testAcc = np.append(testAcc, currTestAcc)

337
338         # Perform optimization routine
339         sess.run(optimizer, feed_dict={X:
340                                         ↳ inputData, Y: inputTarget})

341
342         # Provide info on training progress at the
343         # end of every 10th epoch
344         if currentIter == 0 or currentIter %
345             ↳ (epochSize * 10) == 0:
346             print 'epoch: %5.0f, currTrainErr:
347                   ↳ %10.2f, currValidErr: %10.2f' % \
348                   ((currentIter + 1) / epochSize,
349                     ↳ currTrainErr, currValidErr)

350         #
351             plot_err_graph([trainErr, validErr],
352                           ↳ eta)

353
354         # At the end of each epoch, check for
355         # early stopping possiblity. If so,
356         # break from while loop.

```

```

344         if (currentIter + 1) % epochSize == 0:
345             if should_stop_early(validErr):
346                 writer.close()
347                 print 'Exited from early'
348                 ↳ stopping\n'
349
350             # At every 10 epochs, creates checkpoint
351             ↳ for session
351             #
352             if currentIter % epochSize == 10:
352             #
352                 saver.save(sess, writeDir,
353                 ↳ global_step=currentIter)
353
354             currentIter += 1
355
356             if currentIter == MAX_ITER:
357                 print 'MAX_ITER reached\n'
358                 writer.close()
359
360             # Save session results as dictionary and appends
361             ↳ to summaryList
361             summaryList.append(
362                 {
363                     'eta': eta,
364                     'B': batchSize,
365                     'lambda': lambda_,
366                     'hidden_layers': numHidLayers,
367                     'hidden_units': layerUnits,
368                     'numIter': currentIter + 1 - epochSize,
369                     'epoch': float(currentIter + 1) /
369                         ↳ epochSize - 1,

```

```

370     'trainError':  

371         → trainErr[::epochSize].copy() [::],  

372     'validError':  

373         → validErr[::epochSize].copy() [::],  

374     'testError':  

375         → testErr[::epochSize].copy() [::],  

376     'trainAccuracy':  

377         → trainAcc[::epochSize].copy() [::],  

378     'validAccuracy':  

379         → validAcc[::epochSize].copy() [::],  

380     'testAccuracy':  

381         → testAcc[::epochSize].copy() [::],  

382     'weights_layer_0': weights_10  

383  

384     }  

385  

386     )  

387  

388     # Print stats when a run is done  

389  

390     print 'epochs: %.1f, currentIter: %d' %  

391         → (summaryList[-1]['epoch'],  

392             → summaryList[-1]['numIter'])  

393  

394     print 'trainError: %.3f, validError: %.3f,'  

395         → testError: %.3f' % \  

396             (summaryList[-1]['trainError'][-1],  

397                 → summaryList[-1]['validError'][-1],  

398                 → summaryList[-1]['testError'][-1])  

399  

400     print 'Max trainAccuracy: %.3f, Max validAccuracy:  

401         → %.3f, Max testAccuracy: %.3f' % \  

402             (np.max(trainAcc), \  

403                 np.max(validAcc), \  

404                 np.max(testAcc))
```

```

388     print 'runTime: %3.1fs\n\n' % (time.time() -
389         startTime)
390
390     return summaryList

```

### 3.2.7 Q2.2 :Feedforward fully connected neural network

```

1 summary2_2_1 = train_NN(etaList=[1e-4, 1e-3, 1e-2],
2     numHidLayersList=[1], layerUnitsList=[[1000]],
3     sectionDir='2.2.1')

```

### 3.2.8 Q2.3.1: Number of hidden units

```

1 summary2_3_1 = train_NN(etaList=[1e-3], numHidLayersList=[1,
2     1, 1], layerUnitsList=[[100], [500], [1000]],
3     sectionDir='2.3.1')

```

### 3.2.9 Q2.3.2: Number of layers

```

1 summary2_3_2 = train_NN(etaList=[1e-3], numHidLayersList=[1,
2     2], layerUnitsList=[[1000], [500, 500]],
3     sectionDir='2.3.2')

```

### 3.2.10 Q2.4.1: Dropout

```
1 summary2_4_1 = train_NN(etaList=[1e-3], numHidLayersList=[1],  
2   ↳ layerUnitsList=[[1000]], \  
3     ↳ dropout_rate=0.5, sectionDir='2.4.1')
```

### 3.2.11 Q2.4.2: Visualization

```
1 # Create visualization method  
2 '''  
3 Creates four plots at 25%, 50%, 75% and 100% of training  
4   ↳ epochs.  
5 Each plot consists of 40x25 subplots of 28x28 heatmaps to  
6   ↳ represent each of the 1000x28x28 weights.  
7 '''  
8  
9  
10 def visualise_weights(weights, hasDropout):  
11   # Mark start time  
12   startTime = time.time()  
13  
14   # Calculate total number of epochs ran  
15   numEpochs = weights.shape[2]  
16  
17   # Loop through 25%, 50%, 75%, 100% of runs  
18   for i in range(4):  
19     fig = plt.figure(figsize=(16,20))  
20     # Loop through all 1000 weights in the hidden layer  
21     for j in range(weights.shape[1]):  
22       # Plots on 40x25 subplot  
23       plt.subplot(40, 25, j + 1)
```

```

20
21     # Create 28x28 heatmap of data
22
23         plt.imshow(np.reshape(weights[:, j:j+1, int((i +
24             → 1) * numEpochs / 4) - 1], (28, 28)),
25             → cmap='gray')
26
27     # Turn off axes for each subplot
28
29         plt.gca().axis('off')
30
31
32     # Place subplot main title
33
34         fig.text(0.5, 0.9, str((i + 1) * 100 / 4) + '%',
35             → fontdict=dict(fontsize=20), \
36                 transform=fig.transFigure,
37                 → horizontalalignment='center')
38
39         plt.show()
40
41
42     # Print total plot duration
43
44     print 'Duration: %.1fs' % (time.time() - startTime)
45
46
47     # Visualise weights of model without dropout
48
49     visualise_weights(summary2_2_1[-1]['weights_layer_0'],
50         → hasDropout=False)
51
52
53     # Visualise weights of model with dropout
54
55     visualise_weights(summary2_4_1[-1]['weights_layer_0'],
56         → hasDropout=True)

```

### 3.2.12 Q2.5.1: Random search

```
1 def run_exhaustive_search():
2     """
3         Performs exhaustive search on hyperparameters by sampling
4         from a random uniform distribution.
5         Seed number used is a concatenation of the last two digits
6         of both our student numbers. ('37' + '93')
7
8         Selects random values for:
9             ln(eta): [-7.5, -4.5]
10            hidden layers: [1, 5]
11            hidden layer units: [100, 500]
12            ln(lambda): [-9, -6]
13            dropout: 'yes' or 'no'
14
15
16         # Set random seed
17         np.random.seed(3793)
18
19
20         # Create placeholders to record randomly generated
21         # hyperparameters
22         etaList = []
23         layersList = []
24         unitsList = []
25         lambdaList = []
26         dropoutList = []
27         summary2_5_1 = []
28
29
30         # Assign randomly generated values for model
31         # hyperparameters
```

```

25   for i in range(5):
26       # Randomly sample values for hyperparameters
27       etaList.append(np.exp(np.random.uniform(-7.5, -4.5,
28                           ↳ size=1)))
29       layersList.append(np.random.randint(1, 6, size=1))
30       unitsList.append(np.random.randint(100, 501,
31                           ↳ size=layersList[i]))
32       lambdaList.append(np.exp(np.random.uniform(-9, -6,
33                           ↳ size=1)).astype(np.float32))
34       dropoutList.append(np.random.randint(0, 2))
35
36   # Train each model
37   for i in range(5):
38       # Perform training run
39       summary2_5_1.append(train_NN(etaList[i],
40                                   ↳ layersList[i], [unitsList[i]], \
41                                   ↳ dropout_rate=0.5*dropoutList[i],
42                                   ↳ lambda_=lambdaList[i],
43                                   ↳ sectionDir='2.5.1')[0])
44
45   return summary2_5_1
46
47 summary2_5_1 = run_exhaustive_search()

```

### 3.2.13 Q2.5.2: Exchange ideas among the group

```
1 np.random.seed(1897356)
2 testSummary = train_NN([0.00292], [5], [[496, 276, 486, 299,
→ 376]], sectionDir='Test', dropout_rate=0, lambda_=0.00091)
```