ECE521 Winter 2017: Assignment 2

FuYuan Tee, (999295837) * Chee Loong Soon, (999295793) †

February 27th, 2017

Contents

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

†Equal Contribution (50%), cheeloong.soon@mail.utoronto.ca

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 η

η	*Number of Iterations*	*Validation Loss*	*Best Test Accuracy, %*
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:

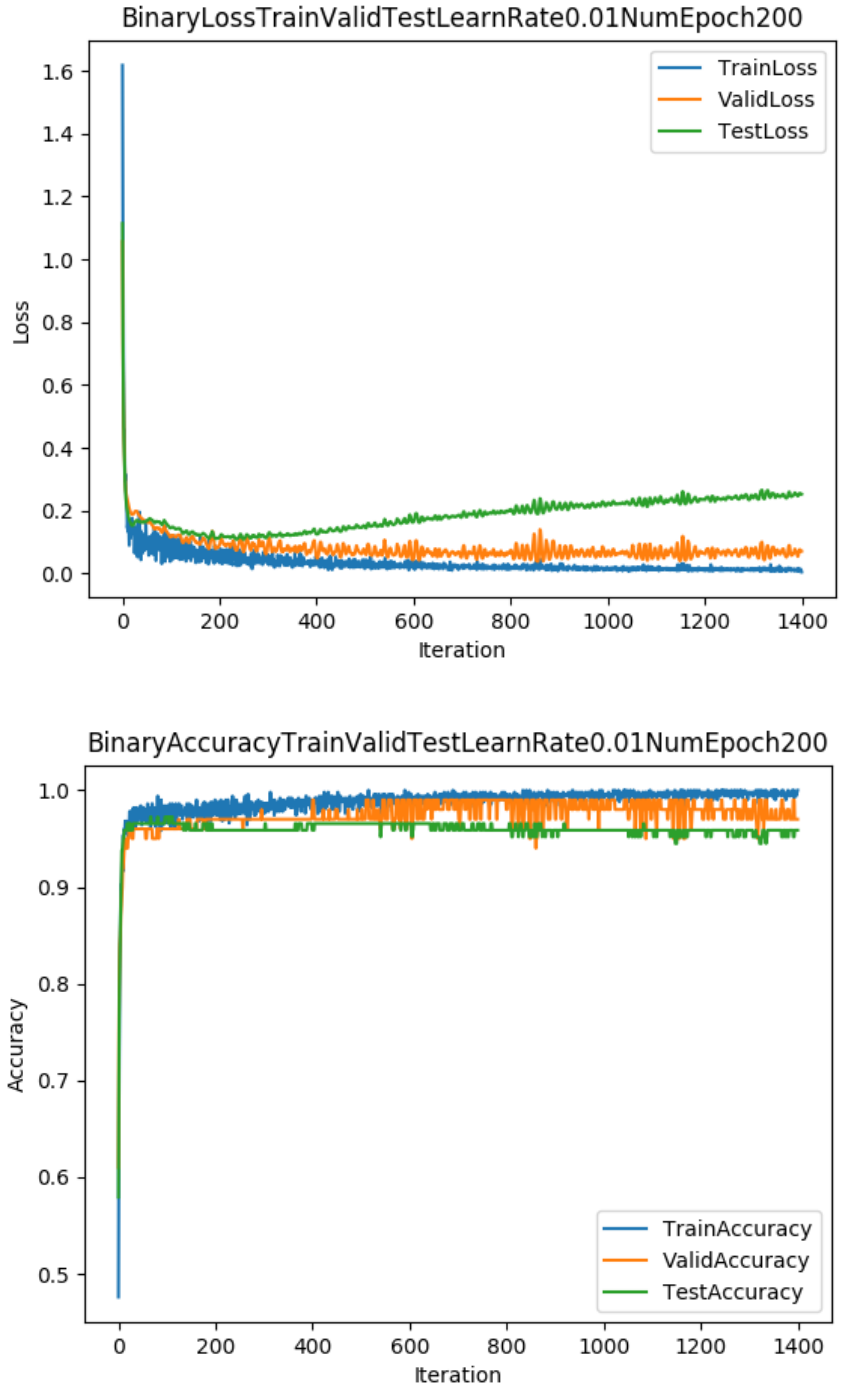


Figure 1: Training and test sets cross-entropy loss and classification accuracy for logistic classifier using stochastic gradient descent (SGD) and best learning rate, $\eta = 0.01$.

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 η

η	*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.

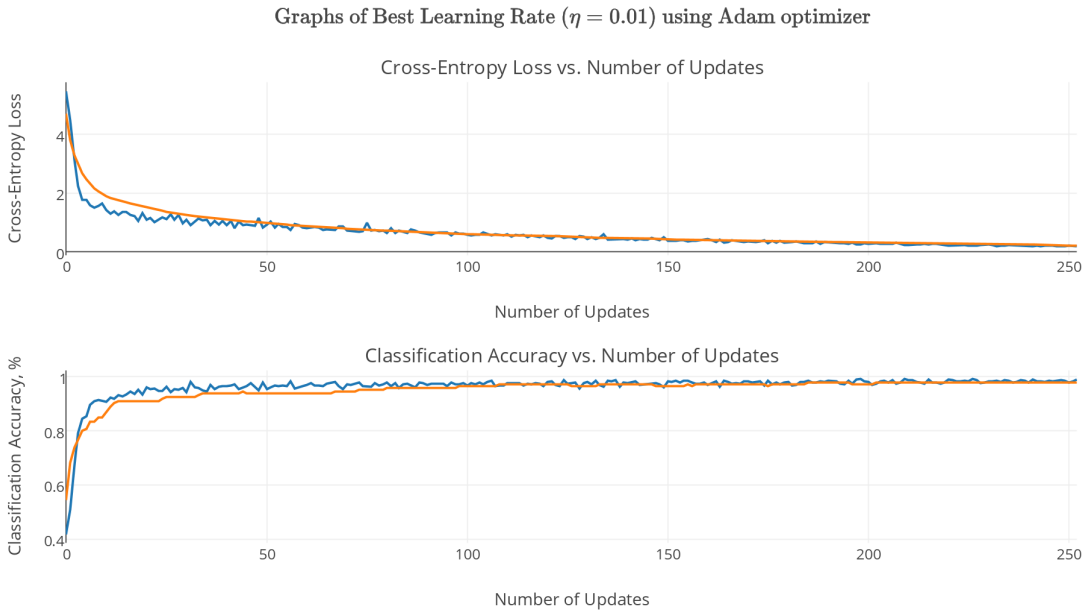


Figure 2: Training and test sets cross-entropy loss and classification accuracy for logistic classifier using Adam optimizer and best learning rate, $\eta = 0.01$.

1.1.3 Comparison with linear regression

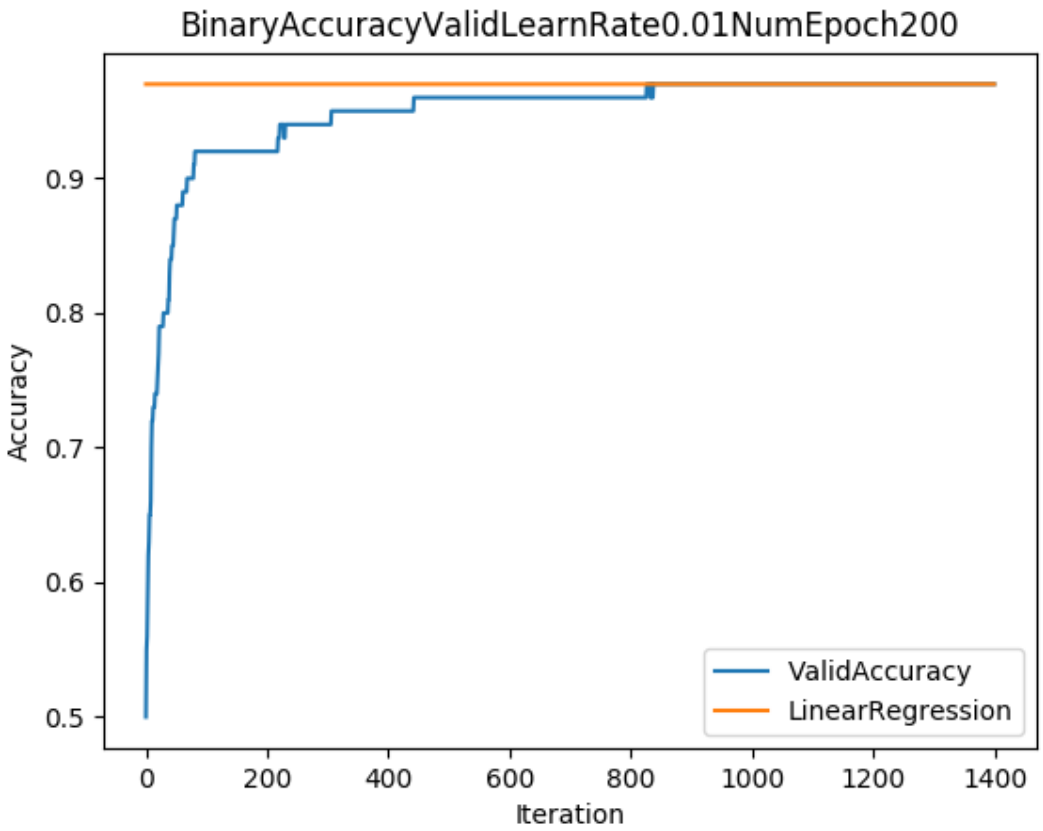


Figure 3: Comparison between Linear Regression Normal equation in yellow and cross-entropy loss in blue for validation classification accuracy

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.

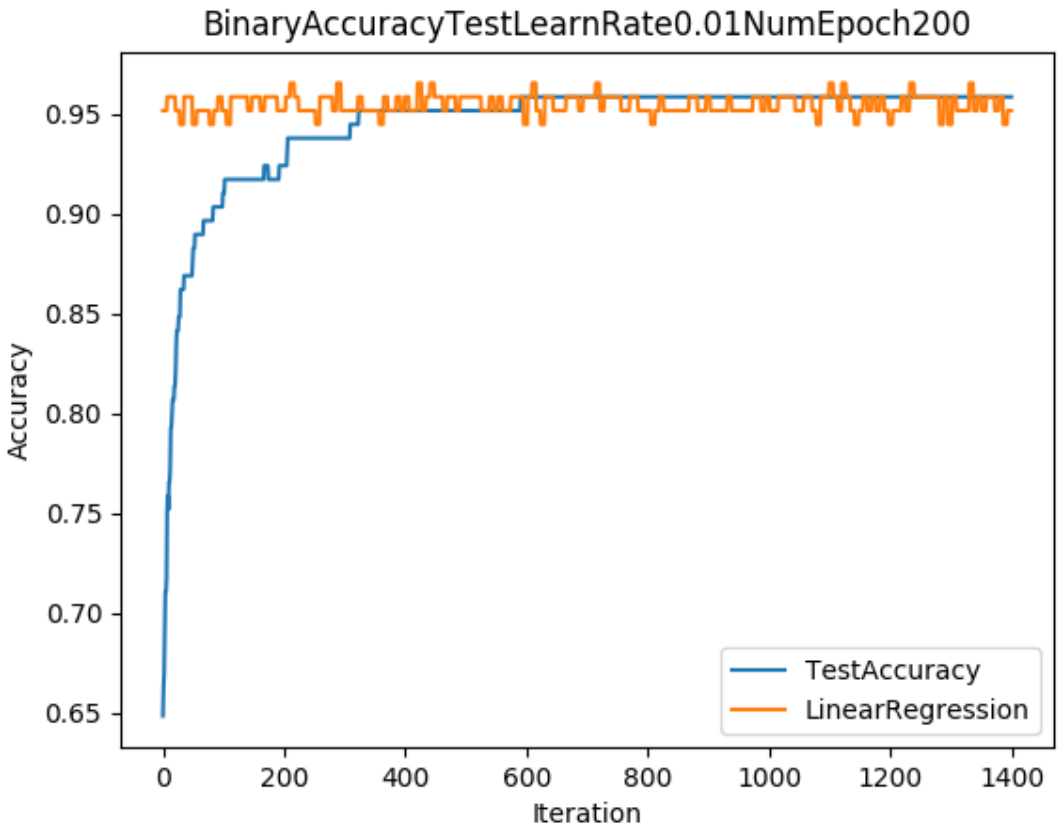


Figure 4: Comparison between Linear Regression Normal equation in yellow and cross-entropy loss in blue for test classification accuracy

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 $\lim_{\hat{y}\to 1} \log(1-\hat{y}) = +\infty$. The fact that the cross-entropy loss penalises misclassifications signficantly more is be vital in minimising errors in classification applications.

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

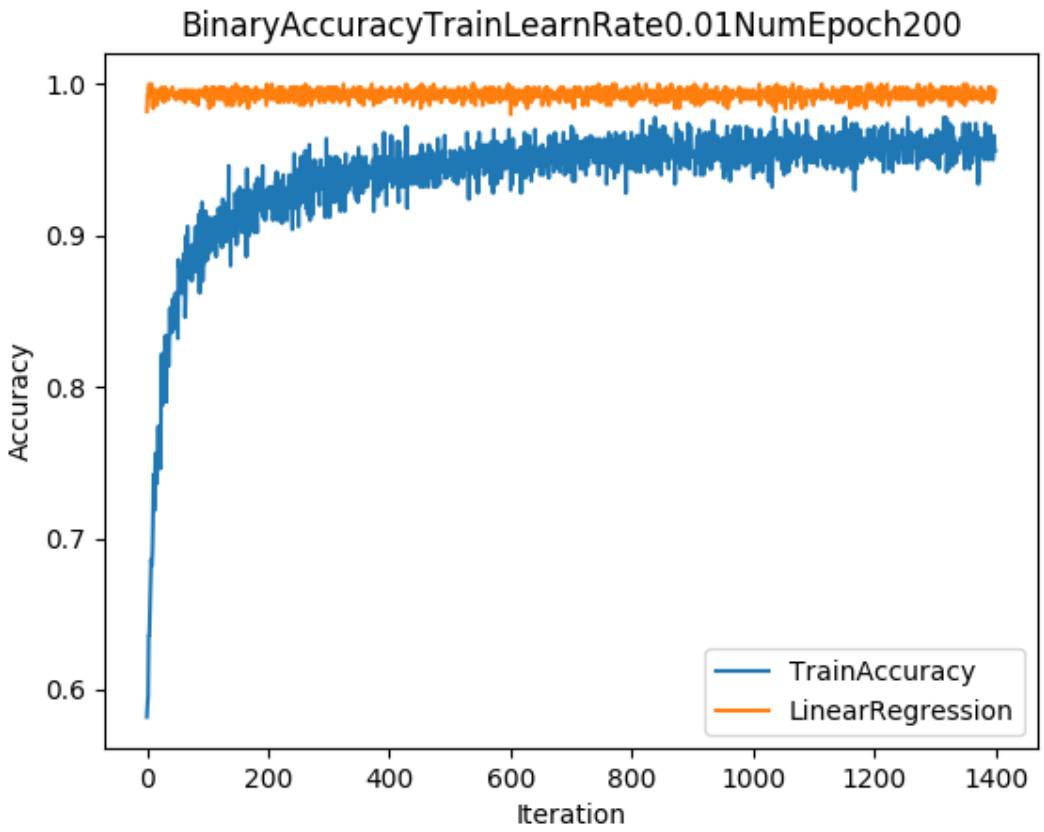


Figure 6: Comparison between Linear Regression Normal equation in yellow and cross-entropy loss in blue for training classification accuracy

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.

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

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

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

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.

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

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

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

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.

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

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.

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

Doing this for every M training case, where m reprents 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 aposterior 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.

$$\mathbf{L} = \begin{bmatrix} 0 & 1 & ... & ... & 1 \\ 1 & 0 & 1 & ... & 1 \\ ... & ... & ... & ... & ... \\ 1 & ... & ... & 1 & 0 \end{bmatrix} \tag{7}$$

If you refer to the proof for section 1.2.2, you will understand that picking the class with largest aposterior 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.

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

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.

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

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

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

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.

$$\arg\max_{j} \Pr\{C_k|\mathbf{x} \in R_j\} \tag{11}$$

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.

$$\mathbf{L} = \begin{bmatrix} L_{11} & L_{12} & ... & L_{1K} \\ L_{21} & L_{22} & ... & L_{2K} \\ ... & ... & ... & ... \\ L_{K1} & ... & ... & L_{KK} \end{bmatrix} \tag{12}$$

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 α. Then, we can re-define this specific region to be the assigned region R_α.

To be clear, we define R_α as the region where all inputs will be classified as class α, where α 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.

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

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.

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

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.

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

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.

$$\mathbf{L}_j = \begin{bmatrix} L_{j1} & L_{j2} & \dots & L_{jK} \end{bmatrix} \tag{16}$$

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

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

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

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

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.

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

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. $L_{ii} = 0 \ \forall i$ is 0 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.

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

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 η

η	*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.

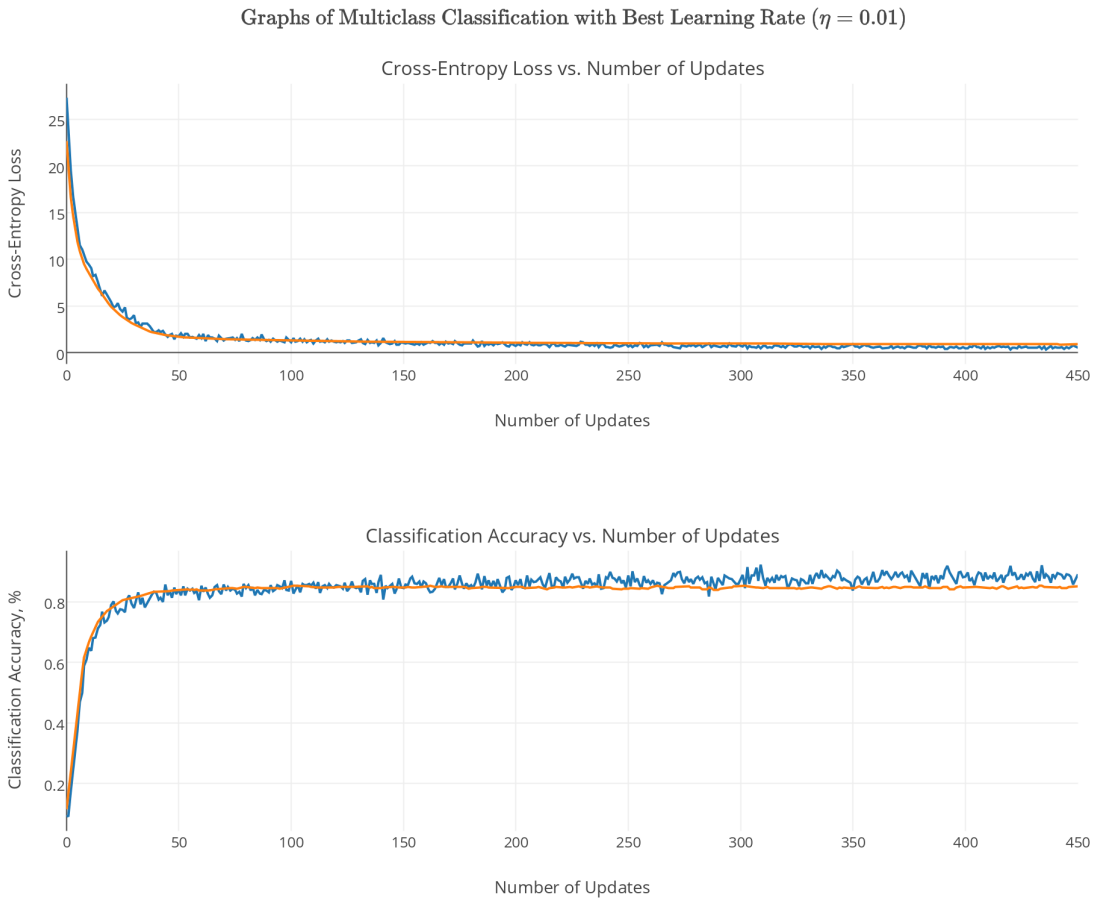


Figure 7: Training and test sets cross-entropy loss and classification accuracy for the softmax classifier with best learning rate, $\eta = 0.01$.

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:

$$\mathcal{L} = -\frac{1}{M}\sum_{m}^{M}\left[t^{(m)}\log y^{(m)} + \left(1-t^{(m)}\right)\log\left(1-y^{(m)}\right)\right] \tag{21}$$

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, $\mathbf{W}$:

$$\begin{aligned}
\frac{\partial\mathcal{L}}{\partial\mathbf{W}} =& \frac{\partial\mathcal{L}}{\partial y}\cdot\frac{\partial y}{\partial\mathbf{z}}\cdot\frac{\partial\mathbf{z}}{\partial\mathbf{W}} \\
=& -\frac{1}{M}\sum_{m}^{M}\left[\frac{t^{(m)}}{y^{(m)}} - \frac{1-t^{(m)}}{1-y^{(m)}}\right]\cdot\left[y^{(m)}\left(1-y^{(m)}\right)\right]\cdot\mathbf{x}^{(m)\prime} \\
=& -\frac{1}{M}\sum_{m}^{M}\left[t^{(m)}\left(1-y^{(m)}\right) - \left(1-t^{(m)}\right)y^{(m)}\right]\cdot\mathbf{x}^{(m)\prime} \\
=& \frac{1}{M}\sum_{m}^{M}\left[y^{(m)} - t^{(m)}\right]\cdot\mathbf{x}^{(m)\prime} \\
=& \frac{1}{M}\sum_{m}^{M}\left[\frac{1}{1+e^{-(\mathbf{W}\prime\mathbf{x}^{(m)})}} - t^{(m)}\right]\cdot\mathbf{x}^{(m)\prime}
\end{aligned} \tag{22}$$

Let the sets $\mathbb{X}_0 = \{\mathbf{x}_0^{(1)},\cdots,\mathbf{x}_0^{(M_0)}\}$ and $\mathbb{X}_1 = \{\mathbf{x}_1^{(1)},\cdots,\mathbf{x}_1^{(M_1)}\}$ 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:

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

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

A dataset is linearly separable when there exists $w_1, \cdots, w_n, k \in \mathbb{R}$ 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:

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

This in turn forces the coefficients of $\mathbf{x}_0^{(m)\prime}$ and $\mathbf{x}_1^{(m)\prime}$ 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:

$$|w_i| = \infty \quad \forall i = 1, \cdots, ..., n \tag{26}$$

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

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 the the target output $\hat{y}$ refers to the predicted output from the sigmoid.

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

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

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

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

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 $\mathbf{x}_0^{(m)'}$ and $\mathbf{x}_1^{(m)'}$ 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, $\|W\|_2 < \infty$.

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 .

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

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

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 $vec\{W\}$ 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 $vec\{W\} = \inf$ (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 $vec\{W\} < \inf$ (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

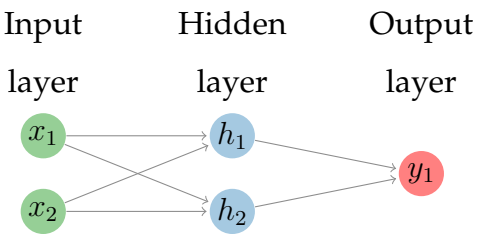
Final output activation has weights 2-Dimension weights and 1-Dimensional bias $\mathbf{W}^3 = [W_1^3, W_2^3]$ and b^3

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 ≤ 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

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

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.

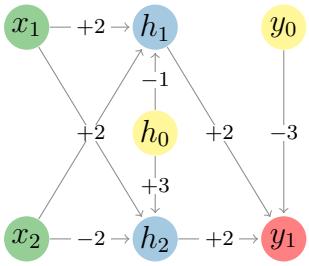
$$\begin{aligned}
\mathbf{W}^1 =& [W_1^1, W_2^1] = [2, 2] \\
b^1 =& -1 \\
\mathbf{W}^2 =& [W_1^2, W_2^2] = [-2, -2] \\
b^2 =& 3 \\
\mathbf{W}^3 =& [W_1^3, W_2^3] = [2, 2] \\
b^3 =& -3
\end{aligned} \tag{32}$$

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.

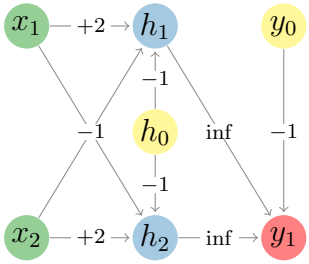
$$
\begin{aligned}
\mathbf{W}^1 =& [W_1^1, W_2^1] = [2, -1] \\
b^1 =& -1 \\
\mathbf{W}^2 =& [W_1^2, W_2^2] = [-1, 2] \\
b^2 =& -1 \\
\mathbf{W}^3 =& [W_1^3, W_2^3] = [\inf, \inf] \\
b^3 =& -1
\end{aligned}
\tag{33}
$$

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

```
    def layerWiseBuildingBlock(self, inputTensor,
     ↪ numberOfHiddenUnits):
        """
        Layer-wise Building Block
        Input:
            inputTensor: Hidden activations from previous
↪ layer
            numberOfHiddenUnitsInCurrentLayer
        Output: z = Weighted sum of the inputs
        Initialize Weight Matrix and Biases in this function
        A list of hidden activations
        No Loops
        """
        numInput = inputTensor.get_shape().as_list()[1]
        numOutput = numberOfHiddenUnits
        # Xavier Initialization
        #variance = tf.div(tf.constant(3.0), tf.add(numInput,
         ↪ tf.constant(numOutput)))
        variance = 3.0/(numInput + numOutput)
        weight =
         ↪ tf.Variable(tf.truncated_normal(shape=[numInput,
         ↪ numOutput], stddev = math.sqrt(variance)))
        bias = tf.Variable(tf.zeros([numOutput]))
        weightedSum = tf.matmul(tf.cast(inputTensor,
         ↪ "float32"), weight) + bias
        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

Hyperparameter	*Value*
Batch size, B	500
L2 regularizer, λ	3×10^{-4}
Hidden Layers	1
Hidden Units	(1000)
Dropout Rate	0%

Table 8: Performance of neural network with respect to η

Learning Rate, η	*Epochs*	*Validation Loss*	*Best Test Accuracy, %*
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.

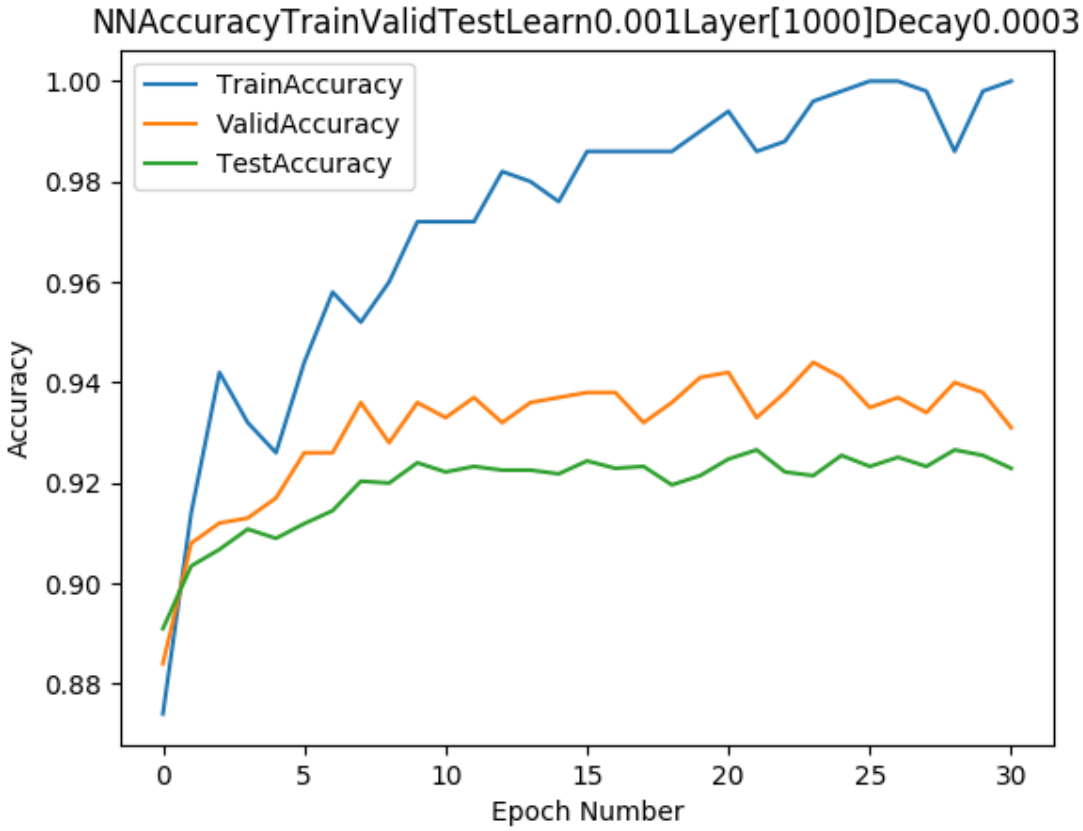


Figure 8: Training, validation and test sets cross-entropy losses and classification errors for neural network with best learning rate, $\eta = 0.001$.

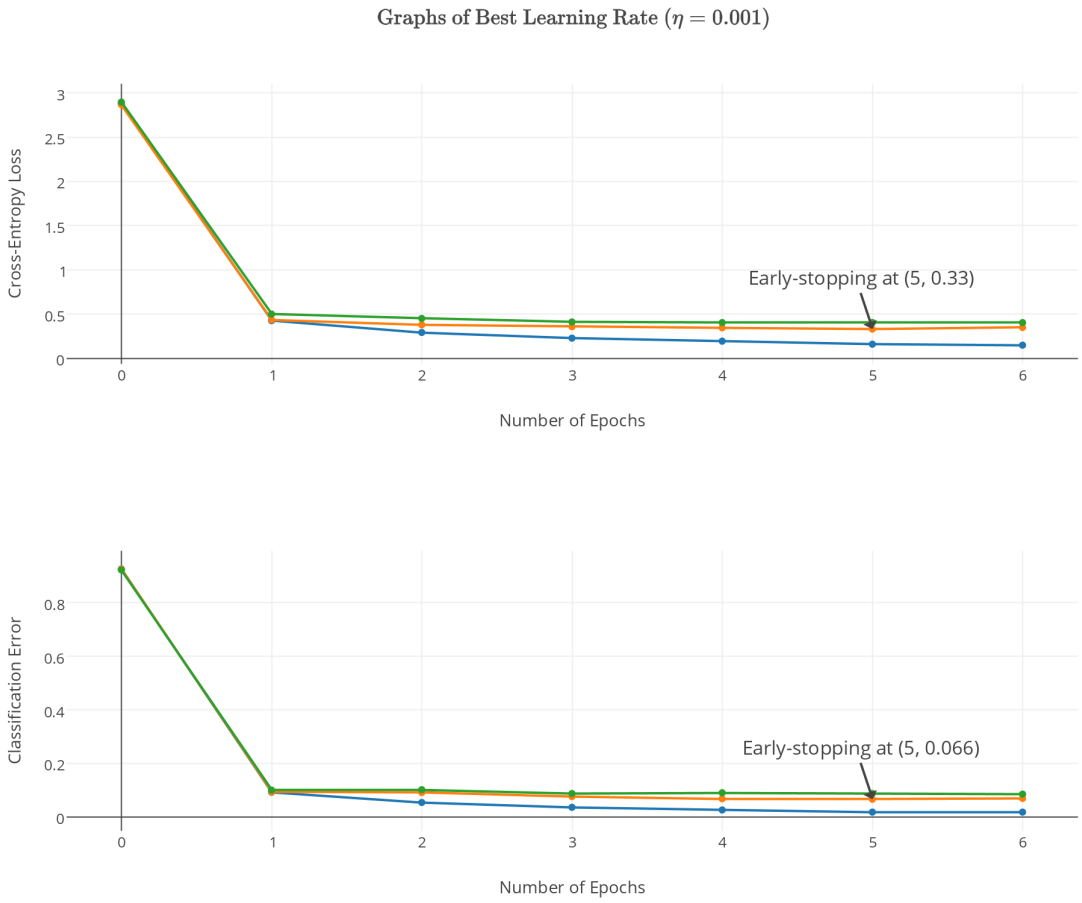


Figure 9: Training, validation and test sets cross-entropy losses and classification errors for neural network with best learning rate, $\eta = 0.001$.

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

Epoch	*Training Error*	*Validation Error*	*Test Error*
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, λ	3×10^{-4}
Learning rate, η	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

Hyperparameter	*Value*
Batch size, B	500
L2 regularizer, λ	3×10^{-4}
Learning rate, η	0.001
Total Hidden Units	1000
Dropout Rate	0%

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

Hidden Layers	*Hidden Units*	*Epochs*	*Validation Loss*	*Best Test Accuracy, %*
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.

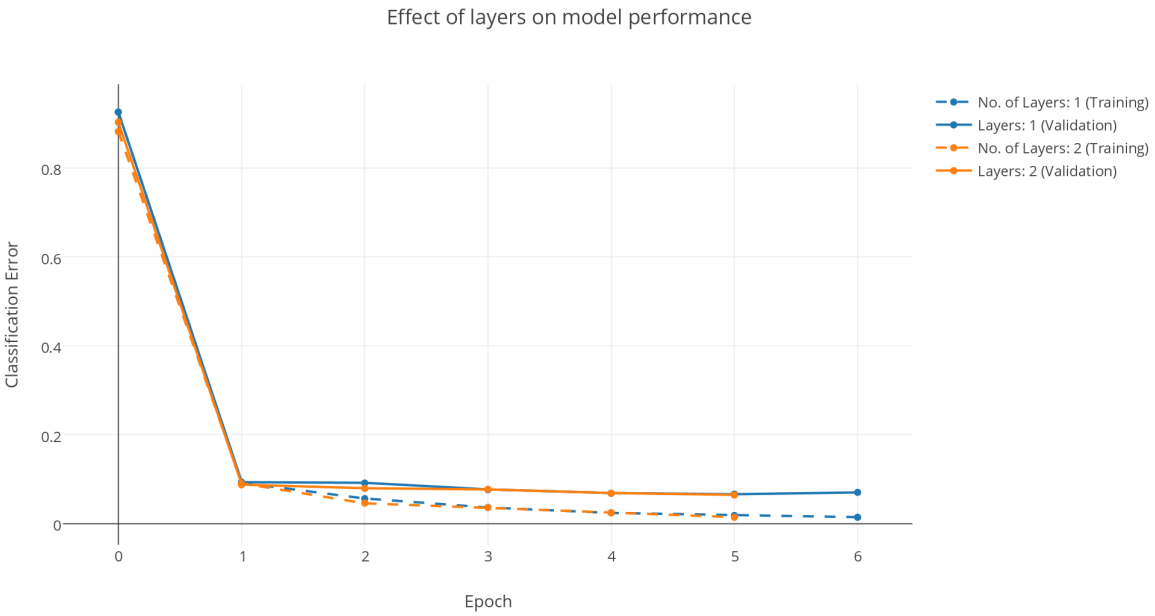


Figure 10: Training and validation classification errors for one-layer and two-layer neural network.

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, λ	3×10^{-4}
Learning rate, η	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.

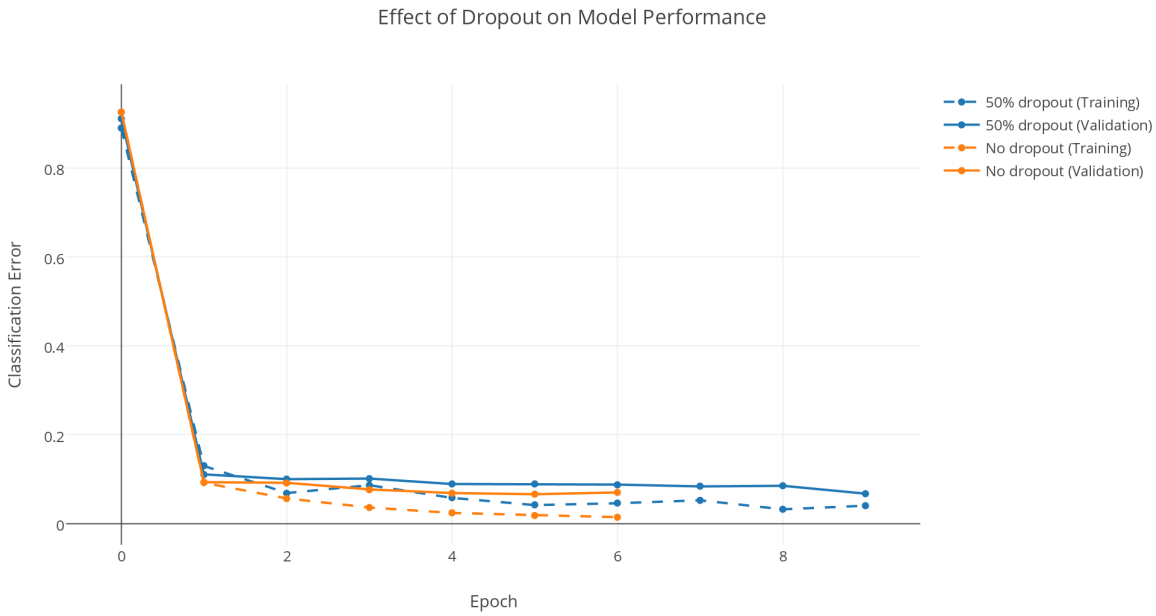


Figure 11: Training and validation classification errors with and without dropout.

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.

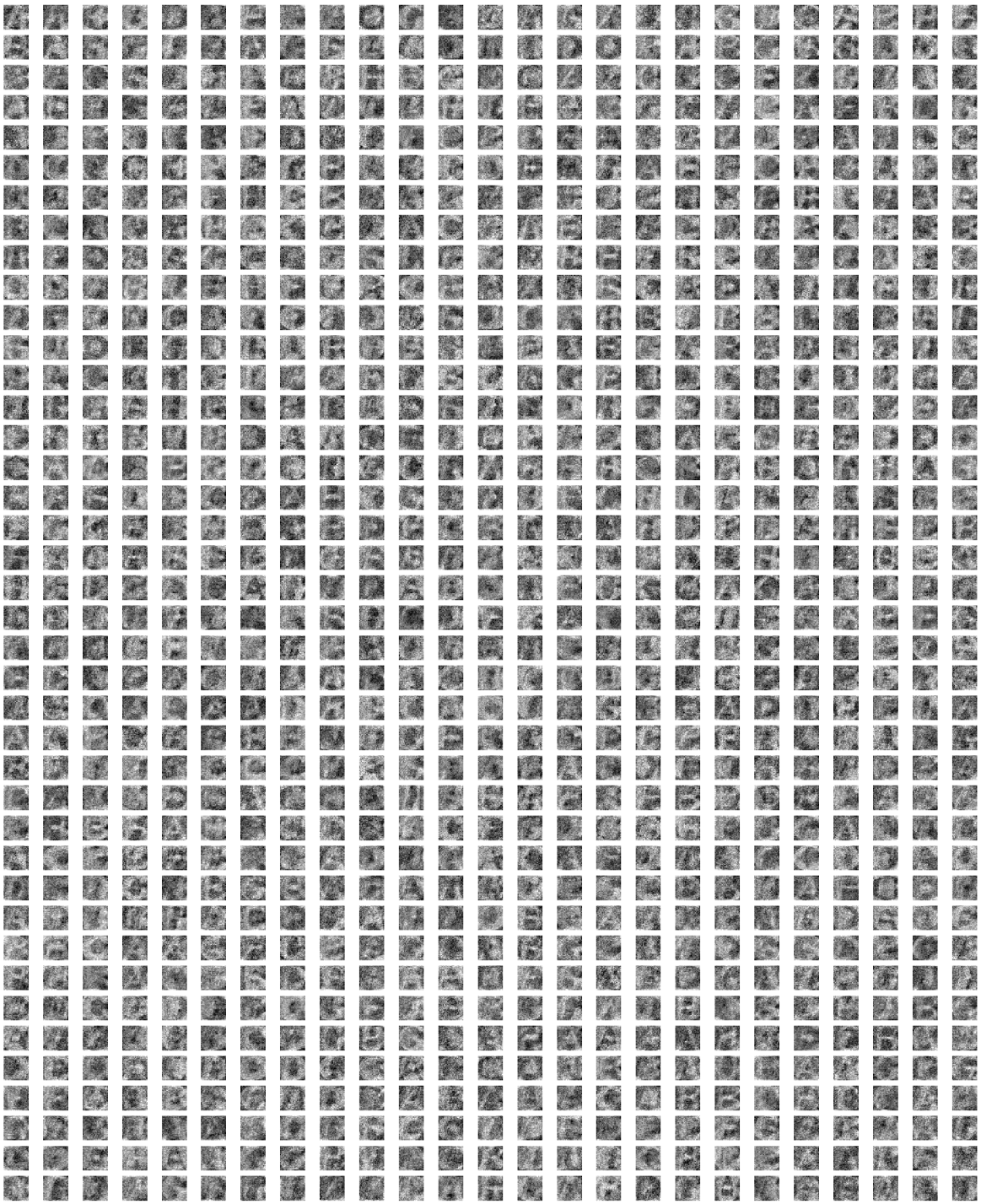
The visualised weights change very quickly from 0% (Gaussian noise) to 25% (distinguishable patterns). After that, the difference between each snapshot of the weights becomes more subtle, in tandem with the plateauing loss function of the model. This trend is observed in both visualisations involving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

One must note that the input data, $\mathbf{x}$, 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.

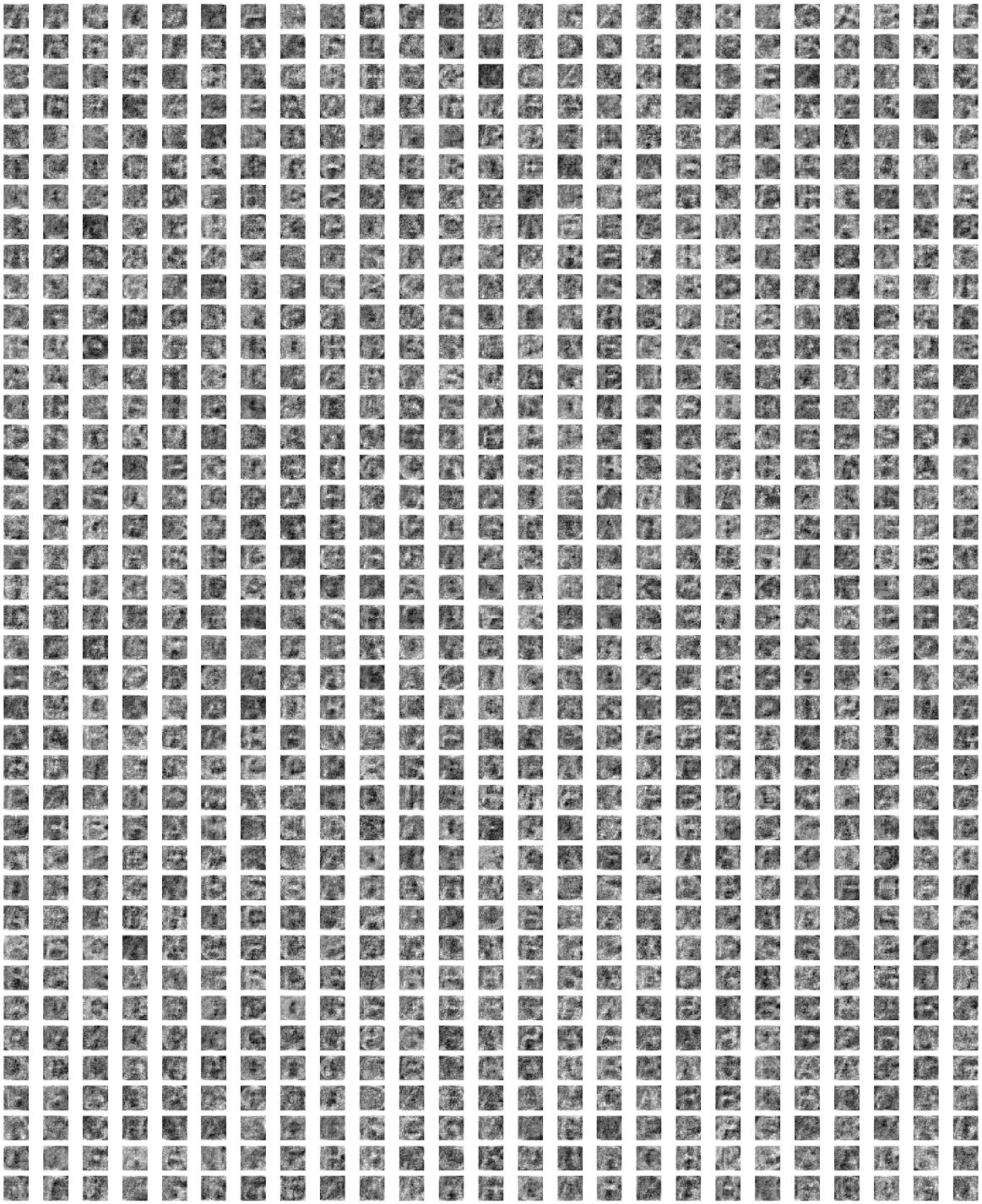
Comparing between the two sets of visualisation with and without dropout, an stark observation is that the patterns are much less distinguishable for the model with dropout. This observation can be explained by the fact that dropout out neurons stochastically trains the neighbouring neurons to learn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

Hidden Layer Weights (without Dropout)

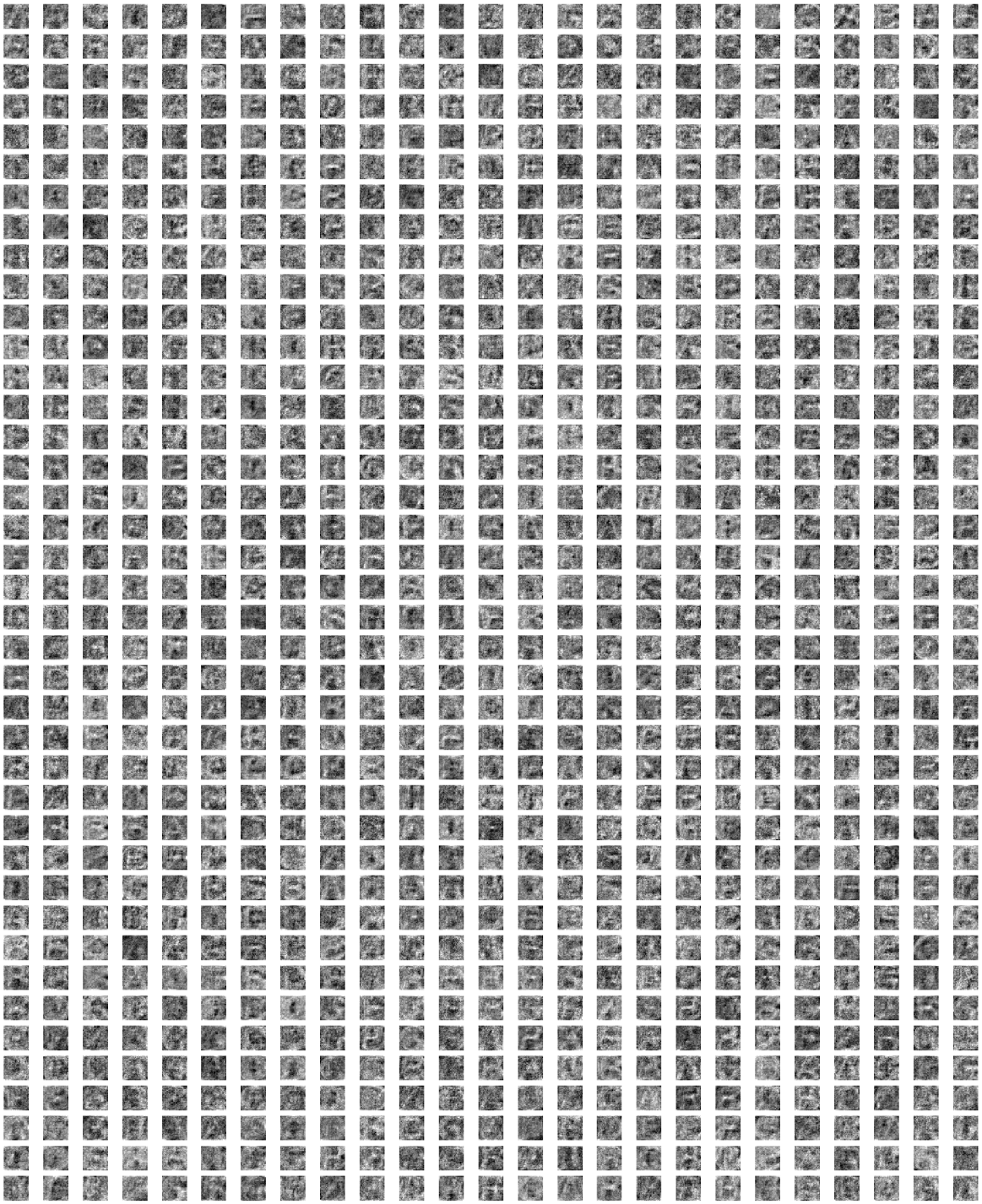
25%



50%



75%



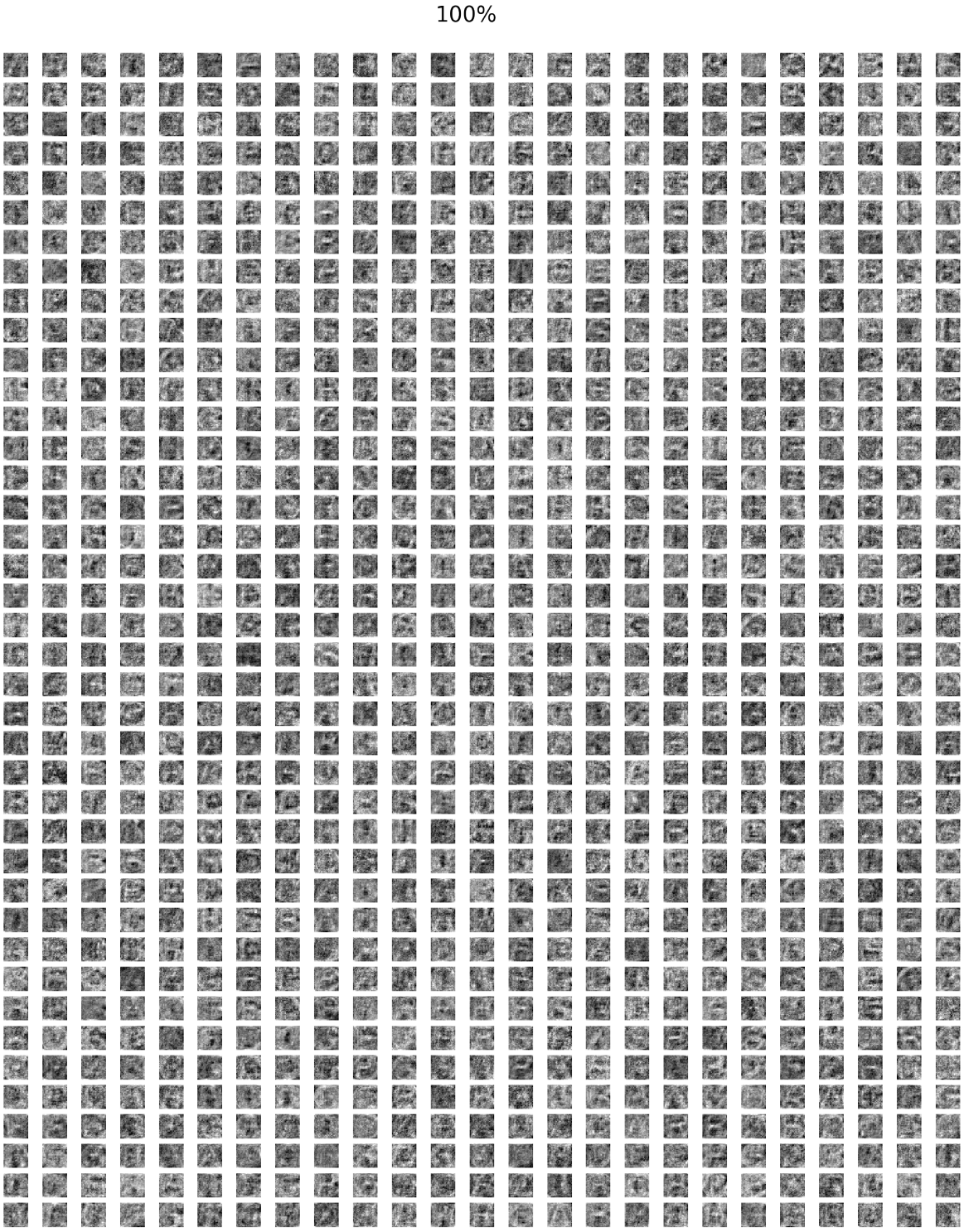
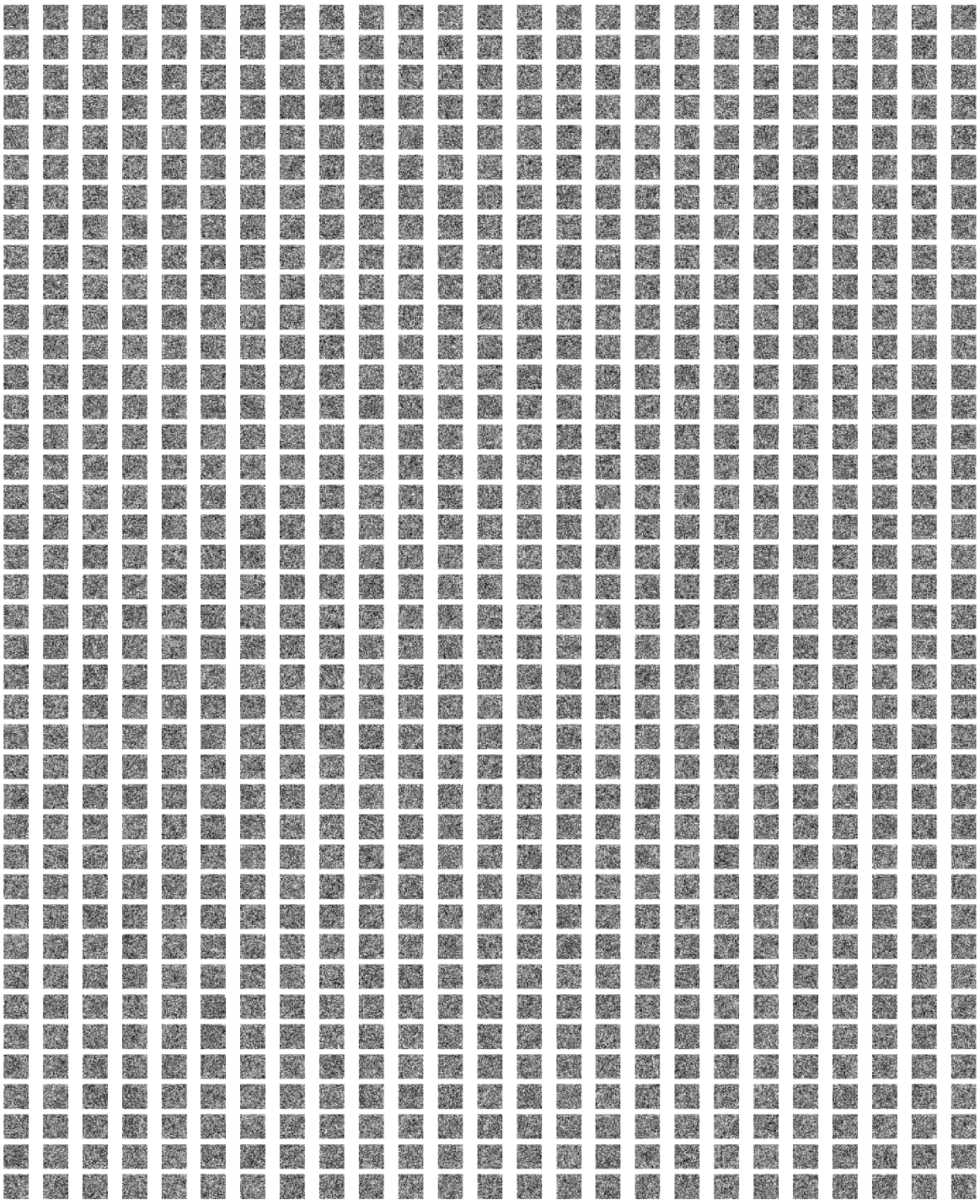


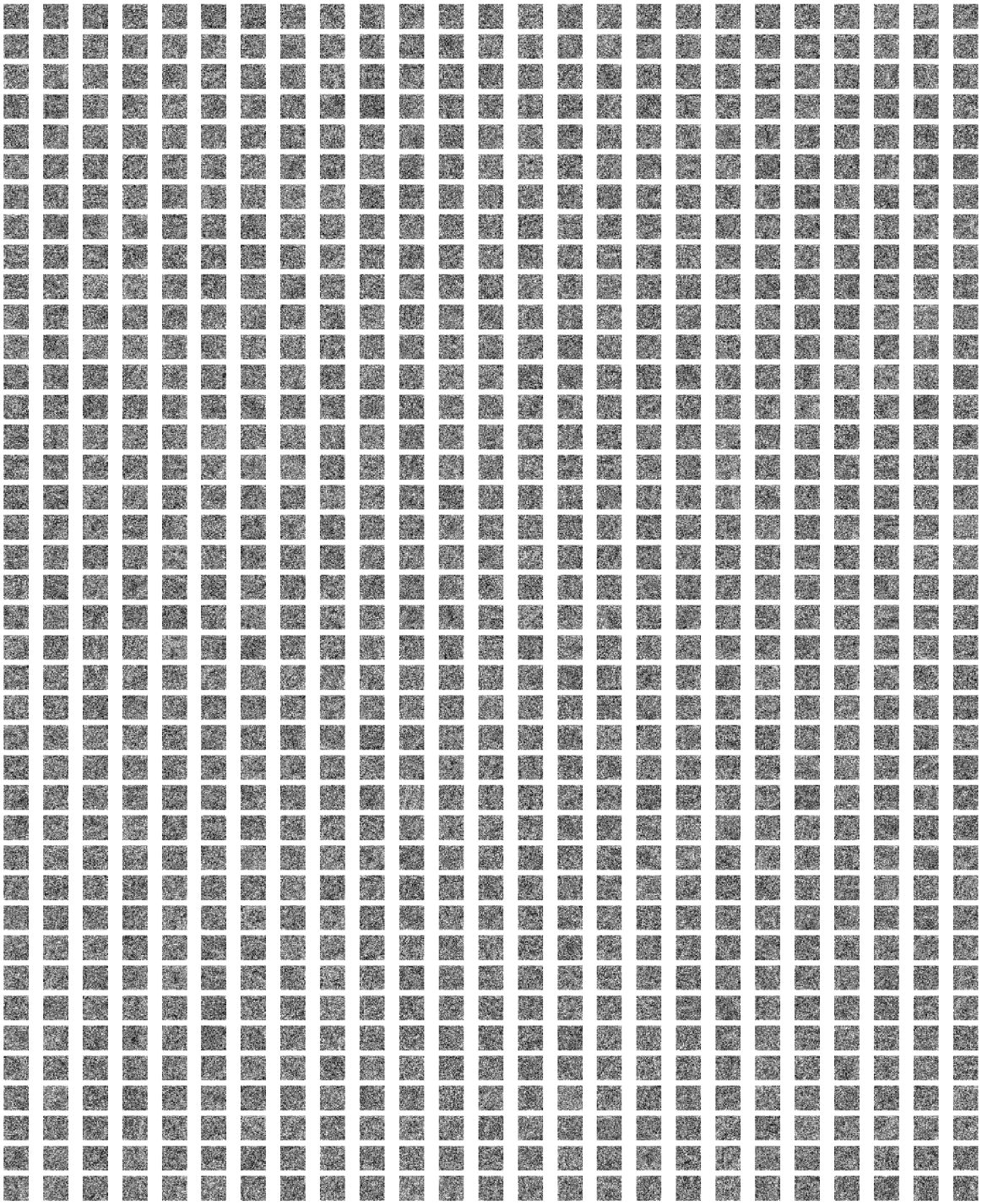
Figure 12: Visualisation of hidden layer weights for model without dropout at 25%, 50%, 75% and 100% of early stopping point.

Hidden Layer Weights (with Dropout)

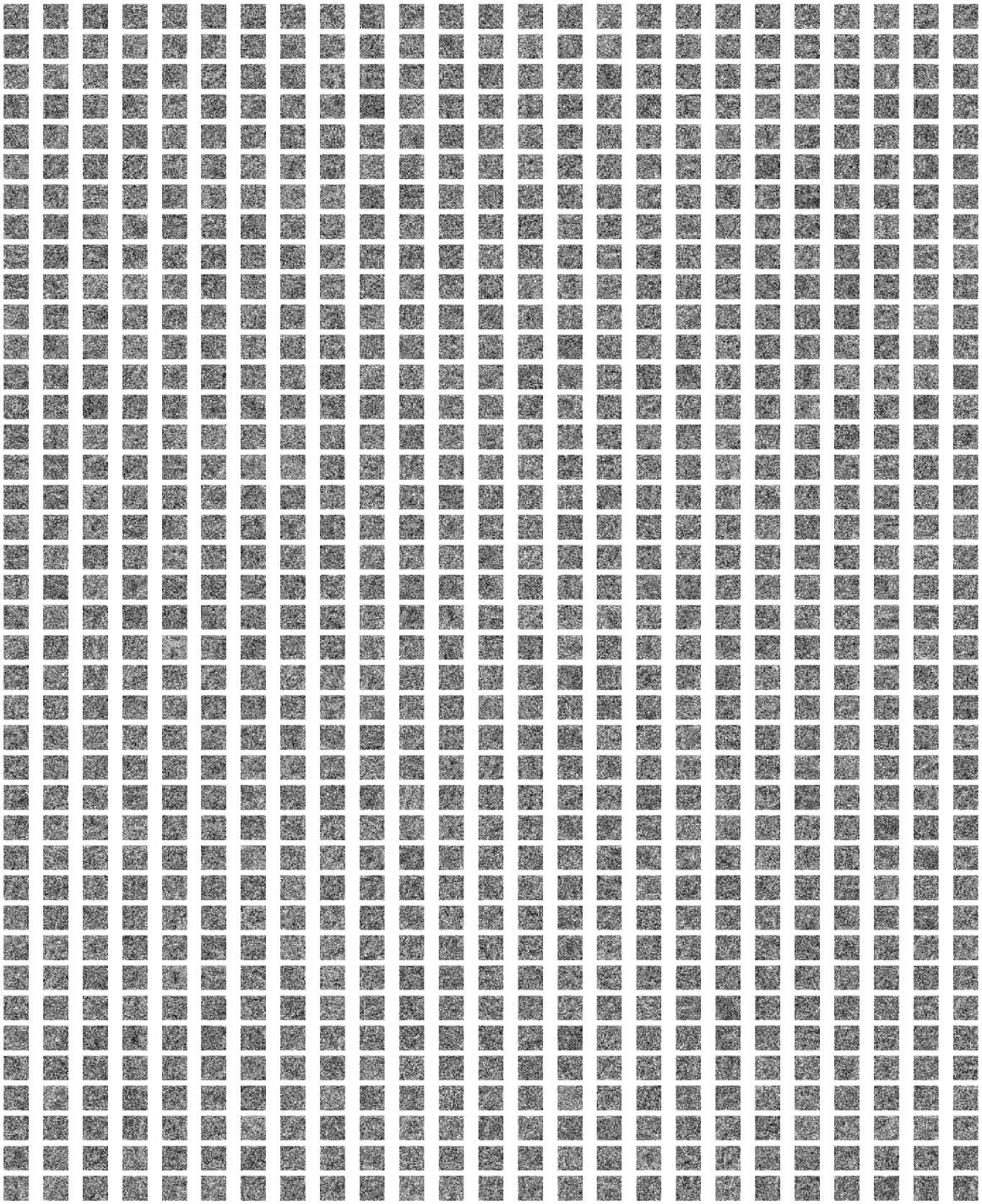
25%



50%



75%



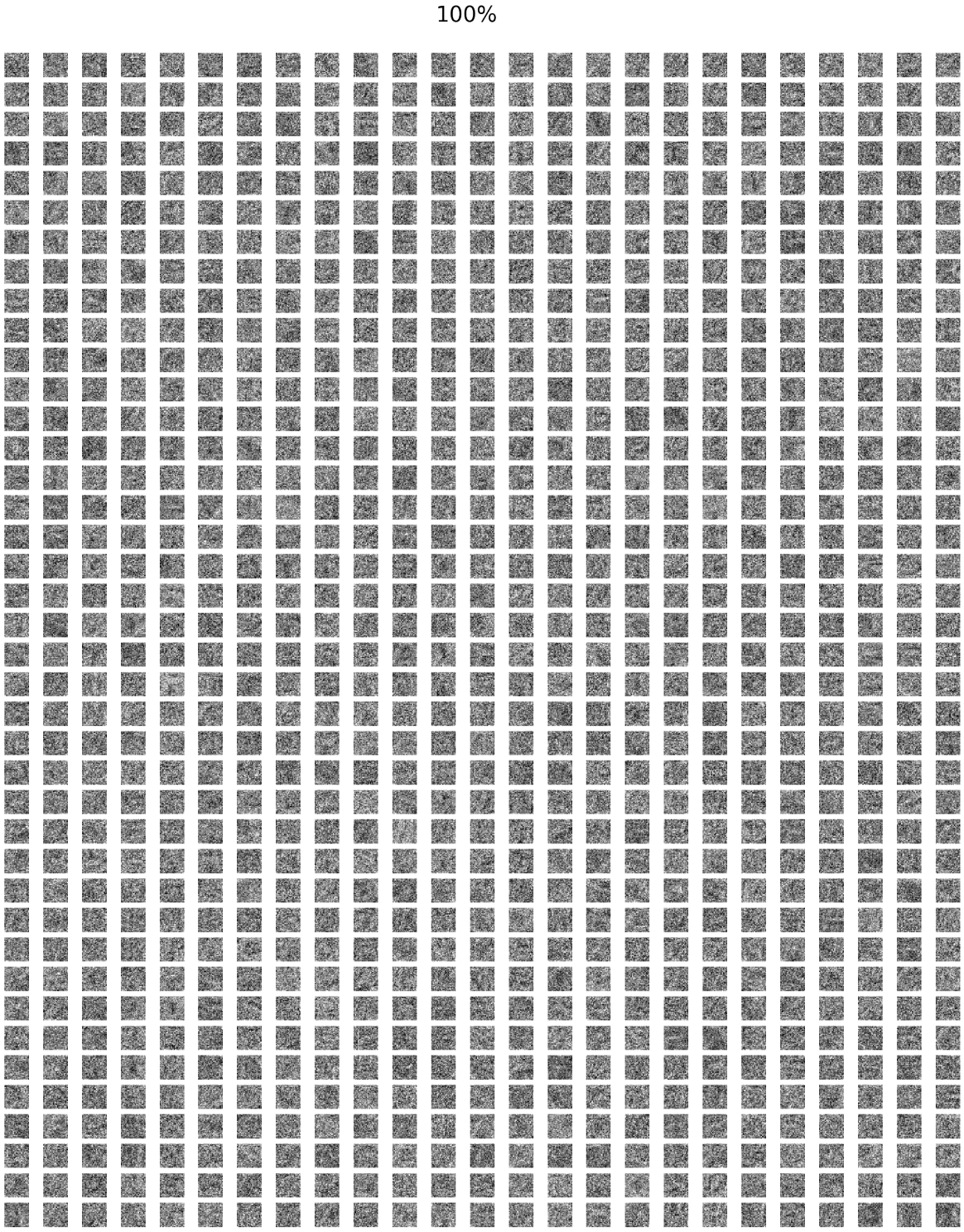


Figure 13: Visualisation of hidden layer weights for model with dropout at 25%, 50%, 75% and 100% of early stopping point.

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, λ	1.70e-3	1.40e-3	6.38e-4	2.67e-4	5.15e-4
Learning rate, η	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.

[1](236, 438, 326, 298, 496)
[2](409, 381, 103, 388)
[3](480, 124, 154, 322, 361)
[4](184, 377)
[5](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[6] hyperparameters on Piazza and its performance

Hyperparameter	*Values*
Batch size, B	500
L2 regularizer, λ	9.10e-4
Learning rate, η	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

[6] 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

```
import datetime
import numpy as np
import tensorflow as tf
import sys
class LogisticRegression(object):
    def __init__(self, trainData, trainTarget, validData,
     ↪ validTarget, testData, testTarget, numEpoch = 1,
     ↪ learningRate = 0.0001, weightDecay = 0.01,
     ↪ optimizerType = "adam", classifierType = "Binary",
     ↪ executeLinearRegression = False, questionTitle = ""):
        self.trainData = trainData
        self.trainTarget = trainTarget
        self.validData = validData
        self.validTarget = validTarget
        self.testData = testData
        self.testTarget = testTarget
        self.miniBatchSize = 500
        self.numEpoch = numEpoch
        self.learningRate = learningRate
        self.weightDecay = weightDecay

```

```
        # To not execute Linear Regression every time
        self.executeLinearRegression = executeLinearRegression
        self.executedLinear = False
        self.WNormalEquationSave = 0
        self.accuracyNormalSave = 0

        self.questionTitle = questionTitle
        self.classifierType = classifierType
        self.optimizer = 0
        if optimizerType == "gd":
            self.optimizer =
             ↪ tf.train.AdamOptimizer(learning_rate =
             ↪ self.learningRate)
        else:
            self.optimizer =
             ↪ tf.train.GradientDescentOptimizer(learning_rate
             ↪ = self.learningRate)
```

```
    def printPlotResults(self, numUpdate, errTrain, accTrain,
     ↪ errValid, errTest, accValid, accTest, xAxis,
     ↪ yTrainErr, yValidErr, yTestErr, yTrainAcc, yValidAcc,
     ↪ yTestAcc, yTrainNormalAcc = 0, yValidNormalAcc = 0,
     ↪ yTestNormalAcc = 0):
        global figureCount
        import matplotlib.pyplot as plt
        print self.classifierType
        print "LearningRate: " , self.learningRate, " Mini
         ↪ batch Size: ", self.miniBatchSize
        print "NumEpoch: ", self.numEpoch
```

```
        print "Iter: ", numUpdate
        print "Final Train MSE: ", errTrain
        print "Final Train Acc: ", accTrain
        print "Final Valid MSE: ", errValid
        print "Final Test MSE: ", errTest
        print "Final Valid Acc: ", accValid
        print "Final Test Acc: ", accTest
        paramStr = "LearnRate"  + str(self.learningRate) +
         ↪  "NumEpoch" + str(self.numEpoch)
        typeLossStr = "Loss"
        typeAccuracyStr = "Accuracy"
        iterationStr = "Iteration"
        trainStr = "Train"
        validStr = "Valid"
        testStr = "Test"
        trainLossStr = trainStr + typeLossStr
        validLossStr = validStr + typeLossStr
        testLossStr = testStr + typeLossStr

        trainAccStr = trainStr + typeAccuracyStr
        validAccStr = validStr + typeAccuracyStr
        testAccStr = testStr + typeAccuracyStr

        linearLabelStr = "LinearRegression"

        figureCount = figureCount + 1
        plt.figure(figureCount)
        title = self.classifierType + typeLossStr + trainStr +
         ↪  paramStr
        plt.title(title)
```

```
35          plt.xlabel(iterationStr)
36          plt.ylabel(typeLossStr)
37          plt.plot(np.array(xAxis), np.array(yTrainErr), label =
             ↪ trainLossStr)
38          plt.legend()
39          plt.savefig(self.questionTitle + title + ".png")
40          plt.close()
41          plt.clf()
42
43          figureCount = figureCount + 1
44          plt.figure(figureCount)
45          title = self.classifierType + typeLossStr + validStr +
             ↪ paramStr
46          plt.title(title)
47          plt.xlabel(iterationStr)
48          plt.ylabel(typeLossStr)
49          plt.plot(np.array(xAxis), np.array(yValidErr), label =
             ↪ validLossStr)
50          plt.legend()
51          plt.savefig(self.questionTitle + title + ".png")
52          plt.close()
53          plt.clf()
54
55          figureCount = figureCount + 1
56          plt.figure(figureCount)
57          title = self.classifierType + typeLossStr + testStr +
             ↪ paramStr
58          plt.title(title)
59          plt.xlabel(iterationStr)
60          plt.ylabel(typeLossStr)
```

```
61          plt.plot(np.array(xAxis), np.array(yTestErr), label =
             ↪ testLossStr)
62          plt.legend()
63          plt.savefig(self.questionTitle + title + ".png")
64          plt.close()
65          plt.clf()
```

```
1           figureCount = figureCount + 1
2           plt.figure(figureCount)
3           title = self.classifierType + typeLossStr + trainStr +
             ↪ validStr + testStr + paramStr
4           plt.title(title)
5           plt.xlabel(iterationStr)
6           plt.ylabel(typeLossStr)
7           plt.plot(np.array(xAxis), np.array(yTrainErr), label =
             ↪ trainLossStr)
8           plt.plot(np.array(xAxis), np.array(yValidErr), label =
             ↪ validLossStr)
9           plt.plot(np.array(xAxis), np.array(yTestErr), label =
             ↪ testLossStr)
10          plt.legend()
11          plt.savefig(self.questionTitle + title + ".png")
12          plt.close()
13          plt.clf()
```

```
1           # Accuracies
2           figureCount = figureCount + 1
3           plt.figure(figureCount)
4           title = self.classifierType + typeAccuracyStr +
             ↪ trainStr + paramStr
```

```
5           plt.title(title)
6           plt.xlabel(iterationStr)
7           plt.ylabel(typeAccuracyStr)
8           plt.plot(np.array(xAxis), np.array(yTrainAcc), label =
             ↪ trainAccStr)
9           accLabel = trainStr + typeAccuracyStr
10          if self.executeLinearRegression:
11              plt.plot(np.array(xAxis),
                 ↪ np.array(yTrainNormalAcc), label =
                 ↪ linearLabelStr)
12          plt.legend()
13          plt.savefig(self.questionTitle + title + ".png")
14          plt.close()
15          plt.clf()
16
17          figureCount = figureCount + 1
18          plt.figure(figureCount)
19          title = self.classifierType + typeAccuracyStr +
             ↪ validStr + paramStr
20          plt.title(title)
21          plt.xlabel(iterationStr)
22          plt.ylabel(typeAccuracyStr)
23          plt.plot(np.array(xAxis), np.array(yValidAcc), label =
             ↪ validAccStr)
24          if self.executeLinearRegression:
25              plt.plot(np.array(xAxis),
                 ↪ np.array(yValidNormalAcc), label =
                 ↪ linearLabelStr)
26          plt.legend()
27          plt.savefig(self.questionTitle + title + ".png")
```

```
28          plt.close()
29          plt.clf()
30
31          figureCount = figureCount + 1
32          title = self.classifierType + typeAccuracyStr +
             ↪ testStr + paramStr
33          plt.figure(figureCount)
34          plt.title(title)
35          plt.xlabel(iterationStr)
36          plt.ylabel(typeAccuracyStr)
37          plt.plot(np.array(xAxis), np.array(yTestAcc), label =
             ↪ testAccStr)
38          if self.executeLinearRegression:
39              plt.plot(np.array(xAxis),
                 ↪ np.array(yTestNormalAcc), label =
                 ↪ linearLabelStr)
40          plt.legend()
41          plt.savefig(self.questionTitle + title + ".png")
42          plt.close()
43          plt.clf()
44
45
46          figureCount = figureCount + 1
47          title = self.classifierType + typeAccuracyStr +
             ↪ trainStr + validStr + testStr + paramStr
48          plt.figure(figureCount)
49          plt.title(title)
50          plt.xlabel(iterationStr)
51          plt.ylabel(typeAccuracyStr)
52          plt.plot(np.array(xAxis), np.array(yTrainAcc), label =
             ↪ trainAccStr)
```

```
        plt.plot(np.array(xAxis), np.array(yValidAcc), label =
         ↪ validAccStr)
        plt.plot(np.array(xAxis), np.array(yTestAcc), label =
         ↪ testAccStr)
        plt.legend()
        plt.savefig(self.questionTitle + title + ".png")
        plt.close()
        plt.clf()

        print self.questionTitle + self.classifierType
        print "Max Test Accuracy is: ",
         ↪ max(np.array(yTestAcc))
```

```
    # Logistic Regression
    def LogisticRegressionMethodBinary(self):
        """
        Implements logistic regression and cross-entropy loss.
        using tf.nn.sigmoid_cross_entropy_with_logits
        Weight-decay coefficient = 0.01
        Mini-batch size = B = 500
        Two Class notMNIST dataset
        Output:
            Plots best training curve
                Cross-entropy Loss vs Number of updates
                Classification Accuracy vs Number of Updates
            Plots best test curve
                Cross-entropy Loss vs Number of updates
                Classification Accuracy vs Number of Updates
            Best Test Classification Accuracy
```

```
        """

        maxTestClassificationAccuracy = 0.0
        W, b, X, y_target, y_predicted, crossEntropyError,
         ↪ train , needTrain, accuracy, WNormalEquation,
         ↪ accuracyNormal, Xall, y_targetAll =
         ↪ self.buildGraphBinary()

        # Session
        init = tf.global_variables_initializer()
        sess = tf.InteractiveSession()
        sess.run(init)
        initialW = sess.run(W)
        initialb = sess.run(b)
        currEpoch = 0
        wList = []
        xAxis = []
        yTrainErr = []
        yValidErr = []
        yTestErr = []
        yTrainAcc = []
        yValidAcc = []
        yTestAcc = []
        yTrainNormalAcc = []
        yValidNormalAcc = []
        yTestNormalAcc = []
        numUpdate = 0
        step = 0
        errTrain = -1
        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([whatYouWantToReturnThatWereReturnedFromBuil
54                  #
                    ↪ feed_dic{valuesToFeedIntoPlaceHoldersThatWereReturned
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:(st
                    ↪ (self.miniBatchSize,784)),Xall:
                    ↪ np.reshape(self.trainData,(3500,784)),
                    ↪ y_targetAll: self.trainTarget, y_target:
```

```
59                  wList.append(currentW)
60                  step = step + 1
61                  xAxis.append(numUpdate)
62                  numUpdate += 1
63                  yTrainErr.append(errTrain)
64
65                  yTrainAcc.append(accTrain)
66                  yTrainNormalAcc.append(accTrainNormal)
67
68                  # These will not optimize the function cause
                     ↪ you did not fetch 'train'
69                  # So it won't have to execute that.
70                  errValid, accValid, accValidNormal =
                     ↪ sess.run([crossEntropyError, accuracy,
                     ↪ accuracyNormal], feed_dict={X:
                     ↪ np.reshape(self.validData,
                     ↪ (self.validData.shape[0],784)), Xall:
                     ↪ np.reshape(self.trainData, (3500, 784)),
                     ↪ y_targetAll: self.trainTarget, y_target:
                     ↪ self.validTarget, needTrain: False})
```

```
                errTest, accTest, accTestNormal =
                 ↪ sess.run([crossEntropyError, accuracy,
                 ↪ accuracyNormal], feed_dict={X:
                 ↪ np.reshape(self.testData,
                 ↪ (self.testData.shape[0], 784)), Xall:
                 ↪ np.reshape(self.trainData, (3500, 784)),
                 ↪ y_targetAll: self.trainTarget, y_target:
                 ↪ self.testTarget, needTrain: False})
                yValidErr.append(errValid)
                yTestErr.append(errTest)
                yValidAcc.append(accValid)
                yValidNormalAcc.append(accValidNormal)
                yTestAcc.append(accTest)
                yTestNormalAcc.append(accTestNormal)
            currEpoch += 1
            logStdOut("e" + str(currEpoch))
        self.printPlotResults(numUpdate, errTrain, accTrain,
         ↪ errValid, errTest, accValid, accTest, xAxis,
         ↪ yTrainErr, yValidErr, yTestErr, yTrainAcc,
         ↪ yValidAcc, yTestAcc, yTrainNormalAcc,
         ↪ yValidNormalAcc, yTestNormalAcc)
```

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

```
        Output:
            Plots best training curve
                Cross-entropy Loss vs Number of updates
                Classification Accuracy vs Number of Updates
            Plots best test curve
                Cross-entropy Loss vs Number of updates
                Classification Accuracy vs Number of Updates
            Best Test Classification Accuracy
        """

        maxTestClassificationAccuracy = 0.0
        W, b, X, y_target, y_predicted, crossEntropyError,
         ↪ train , needTrain, accuracy =
         ↪ self.buildGraphMulti()

        # Session
        init = tf.global_variables_initializer()
        sess = tf.InteractiveSession()
        sess.run(init)
        initialW = sess.run(W)
        initialb = sess.run(b)
        currEpoch = 0
        wList = []
        xAxis = []
        yTrainErr = []
        yValidErr = []
        yTestErr = []
        yTrainAcc = []
        yValidAcc = []
        yTestAcc = []
```

```
        numUpdate = 0
        step = 0
        errTrain = -1
        errValid = -1
        errTest = -1
        accTrain = -1
        accValid = -1
        accTest = -1
        while currEpoch < self.numEpoch:
            self.trainData, self.trainTarget =
             ↪ self.ShuffleBatches(self.trainData,
             ↪ self.trainTarget)
            step = 0
            while step*self.miniBatchSize <
             ↪ self.trainData.shape[0]:
                # train comes from BuildGraph's optimization
                 ↪ method
                # returnedValues =
                 ↪ sess.run([whatYouWantToReturnThatWereReturnedFromBuil
                #
                 ↪ feed_dic{valuesToFeedIntoPlaceHoldersThatWereReturned
                # sess.run() executes whatever graph you built
                 ↪ once up to the point where it needs to
                 ↪ fetch
                # and fetches everything that's in
                 ↪ ([variablesToFetch])
                # Thus, if you don't fetch 'train =
                 ↪ optimizer.minimize(loss)', it won't
                 ↪ optimize it
                _, errTrain, currentW, currentb, yhat,
                 ↪ accTrain= sess.run([train,
                 ↪ crossEntropyError, W, b, y_predicted,
                 ↪ accuracy], feed_dict={X:
                 ↪ np.reshape(self.trainData[step*self.miniBatchSize:(st
                 ↪ (self.miniBatchSize,784)),y_target:
```

```
55                  wList.append(currentW)
56                  step = step + 1
57                  xAxis.append(numUpdate)
58                  numUpdate += 1
59                  yTrainErr.append(errTrain)
60                  yTrainAcc.append(accTrain)
61                  # These will not optimize the function cause
                     ↪ you did not fetch 'train'
62                  # So it won't have to execute that.
63                  errValid, accValid =
                     ↪ sess.run([crossEntropyError, accuracy],
                     ↪ feed_dict={X: np.reshape(self.validData,
                     ↪ (self.validData.shape[0],784)), y_target:
                     ↪ self.validTarget, needTrain: False})
64
65                  errTest, accTest =
                     ↪ sess.run([crossEntropyError, accuracy],
                     ↪ feed_dict={X: np.reshape(self.testData,
                     ↪ (self.testData.shape[0], 784)), y_target:
                     ↪ self.testTarget, needTrain: False})
66                  yValidErr.append(errValid)
67                  yTestErr.append(errTest)
68                  yValidAcc.append(accValid)
69                  yTestAcc.append(accTest)
70              currEpoch += 1
71              logStdOut("e" + str(currEpoch))
72          self.printPlotResults(numUpdate, errTrain, accTrain,
             ↪ errValid, errTest, accValid, accTest, xAxis,
             ↪ yTrainErr, yValidErr, yTestErr, yTrainAcc,
             ↪ yValidAcc, yTestAcc)
```

```
    def buildGraphMulti(self):
        # Parameters to train
        # Images are 28*28 = 784 pixels

        # Multi-class Clasification (10 classes)
        # note: very important to increase dimensions W & b to
         ↪ train each
        # separately using cross entropy error
        W = tf.Variable(tf.truncated_normal(shape=[784, 10],
         ↪ stddev=0.5), name='weights')
        b = tf.Variable(tf.zeros([10]), name='biases')

        # Supervised Inputs
        X = tf.placeholder(tf.float32, [None, 784],
         ↪ name='input_x')

        # y_target for binary class classification
        y_target = tf.placeholder(tf.float32, [None, 10],
         ↪ name='target_y')

        # Label to know if it should train or simply return
         ↪ the errors
        needTrain = tf.placeholder(tf.bool)

        weightDecayCoeff =
         ↪ tf.div(tf.constant(self.weightDecay),tf.constant(2.0))

        # Graph definition
        y_predicted = tf.matmul(X, W) + b

```

```
25          # Multi-class Classification
26          correctPred = tf.equal(tf.argmax(y_predicted, 1),
             ↪ tf.argmax(y_target, 1))
27          accuracy = tf.reduce_mean(tf.cast(correctPred,
             ↪ "float"))
28
29          # Weight Decay Error calculation
30          weightDecayMeanSquareError =
             ↪ tf.reduce_mean(tf.square(W))
31          weightDecayError = tf.multiply(weightDecayCoeff,
             ↪ weightDecayMeanSquareError)
32
33          # Cross Entropy Softmax Error Multi-class
             ↪ Classification
34          # note: Cross entropy only works with values from 0 to
             ↪ 1, so multi-class must be one hot encoded
35          crossEntropySoftmaxError =
             ↪ tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits
             ↪ = y_predicted, labels = y_target))
36          crossEntropySoftmaxError =
             ↪ tf.add(crossEntropySoftmaxError, weightDecayError)
37
38          # Don't train if don't have to for validation and test
             ↪ set
39          #finalTrainingError = tf.select(needTrain,
             ↪ meanSquaredError, tf.constant(0.0))
40          #finalTrainingError = tf.select(needTrain,
             ↪ crossEntropySigmoidError, tf.constant(0.0))
41          finalTrainingError = tf.where(needTrain,
             ↪ crossEntropySoftmaxError, tf.constant(0.0))
```

```
42
43          # Training mechanism
44          #gdOptimizer =
             ↪ tf.train.GradientDescentOptimizer(learning_rate =
             ↪ self.learningRate)
45          adamOptimizer = tf.train.AdamOptimizer(learning_rate =
             ↪ self.learningRate)
46
47          # Train and update the parameters defined
48          # sess.run(train) will execute the optimized function
49          train =
             ↪ adamOptimizer.minimize(loss=finalTrainingError)
50
51          return W, b, X, y_target, y_predicted,
             ↪ crossEntropySoftmaxError, train, needTrain,
             ↪ accuracy
```

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

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

```
32          weightDecayMeanSquareError =
             ↪ tf.reduce_mean(tf.square(W))
33          weightDecayError = tf.multiply(weightDecayCoeff,
             ↪ weightDecayMeanSquareError)
34
35          '''
36          # Mean Square Error Calculation
37          # Divide by 2M instead of M
38          meanSquaredError =
↪ tf.div(tf.reduce_mean(tf.reduce_mean(tf.square(y_predicted
↪ - y_target),
39
↪ reduction_indices=1,
40
↪ name='squared_error'),
41
↪ name='mean_squared_error'), tf.constant(2.0))
42          meanSquaredError = tf.add(meanSquaredError,
↪ weightDecayError)
43          '''
44
45          # Cross Entropy Sigmoid Error Binary-class Calculation
46          crossEntropySigmoidError =
             ↪ tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits
             ↪ = y_predicted, labels = y_target))
47          crossEntropySigmoidError =
             ↪ tf.add(crossEntropySigmoidError, weightDecayError)
48
49          # Don't train if don't have to for validation and test
             ↪ 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
             ↪ was not needed in this assignment
53          #finalTrainingError = tf.where(needTrain,
             ↪ meanSquaredError, tf.constant(0.0))
54
55          finalTrainingError = tf.where(needTrain,
             ↪ crossEntropySigmoidError, tf.constant(0.0))
56
57          # Training mechanism
58
59          # Train and update the parameters defined
60          # sess.run(train) will execute the optimized function
61          train =
             ↪ self.optimizer.minimize(loss=finalTrainingError)
62
63          # Return both errors for plotting for 1.1.3 and set
             ↪ weight to 0 for LogisticRegression for plotting
             ↪ when comparing
64          WNormalEquation = W
65          accuracyNormal = accuracy
66          if self.executeLinearRegression:
67              # 1.1.3 Calculate Linear Regression using Normal
                 ↪ Equation (analytical solution)
68              # Concatenate 1 to account for Bias
69              if self.executedLinear:
70                  WNormalEquation = self.WNormalEquationSave
71                  accuracyNormal = self.accuracyNormalSave
72
```

```
73              else:
74                  OnesForX =
                     ↪ tf.ones(shape=tf.pack([tf.shape(Xall)[0],
                     ↪ 1]))
75                  Xnormal = tf.concat(1,[Xall , OnesForX])
76                  WNormalEquation =
                     ↪ tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(tf.tr
                     ↪ Xnormal)), tf.transpose(Xnormal)),
                     ↪ y_targetAll)
77                  OnesForCurrX =
                     ↪ tf.ones(shape=tf.pack([tf.shape(X)[0],
                     ↪ 1]))
78                  currX = tf.concat(1,[X , OnesForCurrX])
79                  y_predictedNormal = tf.matmul(currX,
                     ↪ WNormalEquation)
80                  correctPredNormal =
                     ↪ tf.equal(tf.cast(tf.greater_equal(y_predictedNormal,
                     ↪ 0.5), tf.float32), tf.floor(y_target))
81                  accuracyNormal =
                     ↪ tf.reduce_mean(tf.cast(correctPredNormal,
                     ↪ "float"))
82                  self.WNormalEquationSave = WNormalEquation
83                  self.accuracyNormalSave = accuracyNormal
84                  self.executedLinear = True
85
86          return W, b, X, y_target, y_predicted,
             ↪ crossEntropySigmoidError, train, needTrain,
             ↪ accuracy, WNormalEquation, accuracyNormal, Xall,
             ↪ y_targetAll
```

```
    def ShuffleBatches(self, trainData, trainTarget):
        # Gets the state as the current time
        rngState = np.random.get_state()
        np.random.shuffle(trainData)
        np.random.set_state(rngState)
        np.random.shuffle(trainTarget)
        return trainData, trainTarget
```

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

```
def ExecuteBinary(questionTitle, numEpoch, learningRates,
 ↪ weightDecay, optimizerType, executeLinearRegression):
    classifierType = "Binary"
    print questionTitle + classifierType
    for learningRate in learningRates:
        with np.load("notMNIST.npz") as data :
            # Need to get new data from scratch everything for
             ↪ a new learning rate
            Data, Target = data ["images"], data["labels"]
            posClass = 2
            negClass = 9
            dataIndx = (Target==posClass) + (Target==negClass)
            Data = Data[dataIndx]/255.
            Target = Target[dataIndx].reshape(-1, 1)
            Target[Target==posClass] = 1
            Target[Target==negClass] = 0
```

```
            np.random.seed(521)
            randIndx = np.arange(len(Data))
            np.random.shuffle(randIndx)
            Data, Target = Data[randIndx], Target[randIndx]
            trainData, trainTarget = Data[:3500],
             ↪ Target[:3500]
            validData, validTarget = Data[3500:3600],
             ↪ Target[3500:3600]
            testData, testTarget = Data[3600:], Target[3600:]
            # Binary Class Linear Regression =  0.001
            # Binary Class Logistic Regression Sigmoid = 0.01
            # Cross entropy loss only works from 0 to 1! Not
             ↪ from 0 to 9 for multi-class
            tf.reset_default_graph()
            l = LogisticRegression(trainData, trainTarget,
             ↪ validData, validTarget, testData, testTarget,
             ↪ numEpoch, learningRate, weightDecay,
             ↪ optimizerType, classifierType,
             ↪ executeLinearRegression, questionTitle)
            l.LogisticRegressionMethodBinary()
            logElapsedTime(questionTitle  + classifierType +
             ↪ str(learningRate))
```

```
def ExecuteMulti(questionTitle, numEpoch, learningRates,
 ↪ weightDecay, optimizerType, executeLinearRegression):
    classifierType = "Multi"
    print questionTitle + classifierType
    with np.load("notMNIST.npz") as data:
        Data, Target = data ["images"], data["labels"]
```

```
        for learningRate in learningRates:
            np.random.seed(521)
            randIndx = np.arange(len(Data))
            np.random.shuffle(randIndx)
            Data = Data[randIndx]/255.
            Target = Target[randIndx]
            trainData, trainTarget = Data[:15000],
             ↪ Target[:15000]
            validData, validTarget = Data[15000:16000],
             ↪ Target[15000:16000]
            # Target values are from 0 to 9
            testData, testTarget = Data[16000:],
             ↪ Target[16000:]
            trainTarget = convertOneHot(trainTarget)
            validTarget = convertOneHot(validTarget)
            testTarget = convertOneHot(testTarget)
            # Multiclass Classification Logistic Regression
             ↪ Softmax = 0.01
            tf.reset_default_graph()
            l = LogisticRegression(trainData, trainTarget,
             ↪ validData, validTarget, testData, testTarget,
             ↪ numEpoch, learningRate, weightDecay,
             ↪ optimizerType, classifierType,
             ↪ executeLinearRegression, questionTitle)
            l.LogisticRegressionMethodMulti()
            logElapsedTime(questionTitle  + classifierType +
             ↪ str(learningRate))
```

```
# Global for logging
questionTitle = "" # Need to be global for logging to work
startTime = datetime.datetime.now()
figureCount = 1 # To not overwrite existing pictures

def logStdOut(message):
    # Temporary print to std out
    sys.stdout = sys.__stdout__
    print message
    # Continue editing same file
    sys.stdout = open("result" + questionTitle + ".txt", "a")

def logElapsedTime(message):
    ''' Logs the elapsedTime with a given message '''
    global startTime
    endTime = datetime.datetime.now()
    elapsedTime = endTime - startTime
    hours, remainder = divmod(elapsedTime.seconds, 3600)
    minutes, seconds = divmod(remainder, 60)
    totalDays = elapsedTime.days
    timeStr = str(message) + ': Days: ' + str(totalDays) +  "
     ↪ hours: " + str(hours) + ' minutes: ' + str(minutes) +
     ↪ ' seconds: ' + str(seconds)
    logStdOut(timeStr)
    startTime = datetime.datetime.now()
```

```
if __name__ == "__main__":
    logStdOut("LogisticRegression")
    # 0.0 LinearRegressionTraining
```

```
    # MeanSquareError learningRate = 0.001, otherwise overshoots

    # 1 Logistic Regression
    # 1.1 Binary Classification, only 2 labels
    # CrossEntropyError, learningRate = 0.01, 98.6% test accuracy highest
    # Default Values
    '''
    numEpoch = 1
    optimizerType = "adam"
    executeLinearRegression = False
    '''
```

```
    '''
    questionTitle = "1.1.1" # Learning with GD
    optimizerType = "gd"
    numEpoch = 201
    learningRates = [0.1, 0.01, 0.001, 0.0001]
    weightDecay = 0.01
    executeLinearRegression = False
    logStdOut("Starting" + questionTitle)
    sys.stdout = open("result" + questionTitle + ".txt", "w") # write a new file from scratch
    ExecuteBinary(questionTitle, numEpoch, learningRates, weightDecay, optimizerType, executeLinearRegression)
    logStdOut("Finished" + questionTitle)
    # '''
```

```
    '''
    questionTitle = "1.1.2" # Beyond plain SGD
    optimizerType = "adam"
    numEpoch = 200
    learningRates = [0.1, 0.01, 0.001, 0.0001]
    weightDecay = 0.01
    executeLinearRegression = False
    logStdOut("Starting" + questionTitle)
    sys.stdout = open("result" + questionTitle + ".txt", "w")
    ExecuteBinary(questionTitle, numEpoch, learningRates,
↪   weightDecay, optimizerType, executeLinearRegression)
    logStdOut("Finished" + questionTitle)
    # '''
```

```
    '''
    # 1.1.3
    questionTitle = "1.1.3" # Comparison With Linear
↪   Regression
    weightDecay = 0.00
    numEpoch = 200
    optimizerType = "adam"
    learningRates = [0.01]
    executeLinearRegression = True
    logStdOut("Starting" + questionTitle)
    sys.stdout = open("result" + questionTitle + ".txt", "w")
    ExecuteBinary(questionTitle, numEpoch, learningRates,
↪   weightDecay, optimizerType, executeLinearRegression)
    logStdOut("Finished" + questionTitle)
    # '''
```

```
    # Multi-class Classification, 10 labels
    # CrossEntropySoftmax Error, learningRate = 0.001
    questionTitle = "1.2.3" # Logistic Regression for
     ↪ multiclass
    weightDecay = 0.01
    numEpoch = 100 # Multi-class Classification
    optimizerType = "adam"
    learningRates = [0.1, 0.01, 0.001]
    executeLinearRegression = False
    logStdOut("Starting" + questionTitle)
    sys.stdout = open("result" + questionTitle + ".txt", "w")
    ExecuteMulti(questionTitle, numEpoch, learningRates,
     ↪ weightDecay, optimizerType, executeLinearRegression)
    logStdOut("Finished" + questionTitle)

\subsubsection{Part2}

import datetime
import numpy as np
import tensorflow as tf
import sys
import math
```

```
class FullyConnectedNeuralNetwork(object):
    """
    Fully Connected Neural Network
    Activation Function: Rectified Linear Unit
    Cost Function: Cross-Entropy
    Output Layer: Softmax
```

```
    Train on entire multi-class MNIST dataset
    Weight initialization using Xavier Initialization
    tf.saver to save model at 25%, 50%, 75% and 100% of
↪   training process
    """

    def __init__(self, trainData, trainTarget, validData,
     ↪ validTarget, testData, testTarget, numEpoch = 1,
     ↪ learningRate = 0.001, weightDecay = 3e-4, doDropout =
     ↪ True, doVisualization = False, hiddenLayers =
     ↪ np.array([1000]), questionTitle=""):
        """
        hiddenLayers is an array indicating the number of
↪   hidden units in each layer
        """
        self.trainData = np.reshape(trainData,
         ↪ (trainData.shape[0], 784))
        self.trainTarget = trainTarget
        self.validData = validData
        self.validTarget = validTarget
        self.testData = testData
        self.testTarget = testTarget

        self.questionTitle = questionTitle
        self.classifierType = "NN"
        self.dropoutProbability = 0.5
        self.doDropout = doDropout
        self.doVisualization = doVisualization

        self.learningRate = learningRate
```

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

```
1       def printPlotResults(self, numUpdate, errTrain, accTrain,
         ↪ errValid, errTest, accValid, accTest, xAxis,
         ↪ yTrainErr, yValidErr, yTestErr, yTrainAcc, yValidAcc,
         ↪ yTestAcc, hiddenImages):
2           global figureCount
3           import matplotlib.pyplot as plt
4           print self.classifierType
5
6           removeFinalLayer = self.hiddenLayers[:-1]
7           print "HiddenLayer: ", str(removeFinalLayer)
8           print "LearningRate: " , self.learningRate, " Mini
             ↪ batch Size: ", self.miniBatchSize
9           print "NumEpoch: ", self.numEpoch
10          print "NumUpdate: ", numUpdate
11          print "Final Train MSE: ", errTrain
12          print "Final Train Acc: ", accTrain
13          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          if self.doDropout:
20              DropoutStr = "Drop"
21          else:
22              DropoutStr = ""
23          removeFinalLayer = self.hiddenLayers[:-1]
24          paramStr = "Learn"  + str(self.learningRate) +
             ↪  DropoutStr + "Layer" + str(removeFinalLayer) +
             ↪  "Decay" + str(self.weightDecay)
25          epochStr = "Epoch" + str(self.numEpoch)
26          typeLossStr = "Loss"
27          typeAccuracyStr = "Accuracy"
28          iterationStr = "Epoch Number"
29          trainStr = "Train"
30          validStr = "Valid"
31          testStr = "Test"
32          trainLossStr = trainStr + typeLossStr
33          validLossStr = validStr + typeLossStr
34          testLossStr = testStr + typeLossStr
35
36          trainAccStr = trainStr + typeAccuracyStr
37          validAccStr = validStr + typeAccuracyStr
38          testAccStr = testStr + typeAccuracyStr
39
40          linearLabelStr = "LinearRegression"
41
```

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

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

```
89          plt.savefig(self.questionTitle + title + epochStr +
             ↪ ".png")
90          plt.close()
91          plt.clf()
92
93          # Accuracies
94          figureCount = figureCount + 1
95          plt.figure(figureCount)
96          title = self.classifierType + typeAccuracyStr +
             ↪ trainStr + paramStr
97          plt.title(title)
98          plt.xlabel(iterationStr)
99          plt.ylabel(typeAccuracyStr)
100         plt.plot(np.array(xAxis), np.array(yTrainAcc), label =
             ↪ trainAccStr)
101         accLabel = trainStr + typeAccuracyStr
102         plt.legend()
103         plt.savefig(self.questionTitle + title + epochStr +
             ↪ ".png")
104         plt.close()
105         plt.clf()
106
107         figureCount = figureCount + 1
108         plt.figure(figureCount)
109         title = self.classifierType + typeAccuracyStr +
             ↪ validStr + paramStr
110         plt.title(title)
111         plt.xlabel(iterationStr)
112         plt.ylabel(typeAccuracyStr)
113         plt.plot(np.array(xAxis), np.array(yValidAcc), label =
             ↪ validAccStr)
```

```
114         plt.legend()
115         plt.savefig(self.questionTitle + title + epochStr +
             ↪ ".png")
116         plt.close()
117         plt.clf()
118 
119         figureCount = figureCount + 1
120         title = self.classifierType + typeAccuracyStr +
             ↪ testStr + paramStr
121         plt.figure(figureCount)
122         plt.title(title)
123         plt.xlabel(iterationStr)
124         plt.ylabel(typeAccuracyStr)
125         plt.plot(np.array(xAxis), np.array(yTestAcc), label =
             ↪ testAccStr)
126         plt.legend()
127         plt.savefig(self.questionTitle + title + epochStr +
             ↪ ".png")
128         plt.close()
129         plt.clf()
130 
131 
132         figureCount = figureCount + 1
133         title = self.classifierType + typeAccuracyStr +
             ↪ trainStr + validStr + testStr + paramStr
134         plt.figure(figureCount)
135         plt.title(title)
136         plt.xlabel(iterationStr)
137         plt.ylabel(typeAccuracyStr)
138         plt.plot(np.array(xAxis), np.array(yTrainAcc), label =
             ↪ trainAccStr)
```

```
139         plt.plot(np.array(xAxis), np.array(yValidAcc), label =
             ↪ validAccStr)
140         plt.plot(np.array(xAxis), np.array(yTestAcc), label =
             ↪ testAccStr)
141         plt.legend()
142         plt.savefig(self.questionTitle + title + epochStr +
             ↪ ".png")
143         plt.close()
144         plt.clf()
145
146         print self.questionTitle + self.classifierType
147         print "Max Test Accuracy is: ",
             ↪ max(np.array(yTestAcc))
148
149         if self.doVisualization:
150             figureCount = figureCount + 1
151             plt.figure(figureCount)
152             # Average across the 1000 hidden units to get a
                 ↪ grascale image
153             # To convert from (784,1000) to (784) to (28,28)
154             hiddenImageToPlot =
                 ↪ np.reshape(np.average(hiddenImages[0], 1),
                 ↪ (28,28))
155             plt.imshow(hiddenImageToPlot,
                 ↪ interpolation="nearest", cmap="gray")
156             plt.savefig("FirstHiddenLayerAverageEpoch" +
                 ↪ str(self.numEpoch) + ".png")
157             logStdOut('Done Plotting AverageImage')
158
159             figureCount = figureCount + 1
```

```
            plt.figure(figureCount)
            numHiddenLayer = hiddenImages[0].shape[1]
            fig = plt.figure(figsize=(28,28))
            numCol = 25
            numRow = numHiddenLayer/numCol
            for eachHiddenLayer in xrange(numHiddenLayer):
                logStdOut('hl:' + str(eachHiddenLayer))
                # Draw each figures (28, 28)
                currImage = hiddenImages[0]
                hiddenImageToPlot =
                 ↪ np.reshape(currImage[::,eachHiddenLayer:eachHiddenLay
                 ↪ (28,28))
                ax = fig.add_subplot(numCol, numRow,
                 ↪ eachHiddenLayer+1)
                ax.imshow(hiddenImageToPlot,
                 ↪ interpolation="nearest", cmap="gray")
                ax.xaxis.set_visible(False)
                ax.yaxis.set_visible(False)
                x0, x1 = ax.get_xlim()
                y0, y1 = ax.get_ylim()
                ax.set_aspect(abs(x1-x0)/abs(y1-y0))
            logStdOut('Saving all images')
            plt.savefig("FirstHiddenLayerAllUnitsEpoch" +
             ↪ str(self.numEpoch) + ".png")
            logStdOut('Saved all images')
```

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

```
        np.random.shuffle(trainData)
        np.random.set_state(rngState)
        np.random.shuffle(trainTarget)
        return trainData, trainTarget
```

```
    def layerWiseBuildingBlock(self, inputTensor,
     ↪ numberOfHiddenUnits):
        """
        Layer-wise Building Block
        Input:
            inputTensor: Hidden activations from previous
↪ layer
            numberOfHiddenUnitsInCurrentLayer
        Output: z = Weighted sum of the inputs
        Initialize Weight Matrix and Biases in this function
        A list of hidden activations
        No Loops
        """
        numInput = inputTensor.get_shape().as_list()[1]
        numOutput = numberOfHiddenUnits
        # Xavier Initialization
        #variance = tf.div(tf.constant(3.0), tf.add(numInput,
         ↪ tf.constant(numOutput)))
        variance = 3.0/(numInput + numOutput)
        weight =
         ↪ tf.Variable(tf.truncated_normal(shape=[numInput,
         ↪ numOutput], stddev = math.sqrt(variance)))
        bias = tf.Variable(tf.zeros([numOutput]))
        weightedSum = tf.matmul(tf.cast(inputTensor,
         ↪ "float32"), weight) + bias
```

```
20
21          tf.add_to_collection("Weights", weight)
22
23          # Add to collection for weight decay loss
24          weightDecayCoeff =
             ↪ tf.div(tf.cast(tf.constant(self.weightDecay),
             ↪ "float32"),tf.constant(2.0))
25          # Weight Decay Error calculation
26          weightDecayMeanSquareError =
             ↪ tf.reduce_mean(tf.square(weight))
27          weightDecayError = tf.multiply(weightDecayCoeff,
             ↪ weightDecayMeanSquareError)
28
29          tf.add_to_collection("WeightDecayLoss",
             ↪ weightDecayError)
30
31          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],
             ↪ name='target_y')
6           inputTensor = X
7
8           for currLayer in self.hiddenLayers:
9               weightedSum =
                 ↪ self.layerWiseBuildingBlock(inputTensor,
                 ↪ currLayer)
```

```
            # Parse with activation function of ReLu
            inputTensor = tf.nn.relu(weightedSum)
            # 2.4.1 Dropout
            if self.doDropout:
                inputTensor = tf.nn.dropout(inputTensor,
                 ↪ self.dropoutProbability)

        # inputTensor is now the final hidden layer, but only
         ↪ need weighted Sum
        # Need add one more with softmax for output
        y_predicted = weightedSum

        # Multi-class Classification
        correctPred = tf.equal(tf.argmax(y_predicted, 1),
         ↪ tf.argmax(y_target, 1))
        accuracy = tf.reduce_mean(tf.cast(correctPred,
         ↪ "float"))


        # Cross Entropy Softmax Error Multi-class
         ↪ Classification
        # note: Cross entropy only works with values from 0 to
         ↪ 1, so multi-class must be one hot encoded
        crossEntropySoftmaxError =
         ↪ tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits
         ↪ labels=y_target))
        adamOptimizer = tf.train.AdamOptimizer(learning_rate =
         ↪ self.learningRate)

        # Calculate weight decay error
```

```
        totalWeightDecayError =
         ↪ sum(tf.get_collection("WeightDecayLoss"))
        crossEntropySoftmaxError =
         ↪ tf.add(crossEntropySoftmaxError,
         ↪ totalWeightDecayError)
        finalTrainingError = crossEntropySoftmaxError
        train =
         ↪ adamOptimizer.minimize(loss=finalTrainingError)

        weights = tf.get_collection("Weights")

        return X, y_target, y_predicted, finalTrainingError,
         ↪ train, accuracy, weights
```

```
    def NeuralNetworkMethod(self):
        maxTestClassificationAccuracy = 0.0
        inputTensor = tf.stack(self.trainData)
        # Build the fully connected Neural Network
        X, y_target, y_predicted, crossEntropyError, train,
         ↪ accuracy, weights =
         ↪ self.buildFullyConnectedNeuralNetwork()

        # Session
        init = tf.global_variables_initializer()
        sess = tf.InteractiveSession()
        sess.run(init)
        currEpoch = 0
        xAxis = []
        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%
             ↪ from early stopping point
28          while currEpoch <= self.numEpoch:
29              self.trainData, self.trainTarget =
                 ↪ self.ShuffleBatches(self.trainData,
                 ↪ self.trainTarget)
30              step = 0
31              while step*self.miniBatchSize <
                 ↪ self.trainData.shape[0]:
32                  # train comes from BuildGraph's optimization
                     ↪ method
33                  # returnedValues =
                     ↪ sess.run([whatYouWantToReturnThatWereReturnedFromBuil
34                  #
                     ↪ feed_dic{valuesToFeedIntoPlaceHoldersThatWereReturned
35                  # sess.run() executes whatever graph you built
                     ↪ once up to the point where it needs to
                     ↪ fetch
```

```
36                  # and fetches everything that's in
                     ↪ ([variablesToFetch])
37                  # Thus, if you don't fetch 'train =
                     ↪ optimizer.minimize(loss)', it won't
                     ↪ optimize it
38                  _, errTrain, yhat, accTrain, hiddenImages =
                     ↪ sess.run([train, crossEntropyError,
                     ↪ y_predicted, accuracy, weights],
                     ↪ feed_dict={X:
                     ↪ np.reshape(self.trainData[step*self.miniBatchSize:(st
                     ↪ (self.miniBatchSize,784)),y_target:
                     ↪ self.trainTarget[step*self.miniBatchSize:(step+1)*sel
39                  step = step + 1
40                  numUpdate += 1
41                  # These will not optimize the function cause
                     ↪ you did not fetch 'train'
42                  # So it won't have to execute that.
43                  errValid, accValid =
                     ↪ sess.run([crossEntropyError, accuracy],
                     ↪ feed_dict={X: np.reshape(self.validData,
                     ↪ (self.validData.shape[0],784)), y_target:
                     ↪ self.validTarget})
44
45                  errTest, accTest =
                     ↪ sess.run([crossEntropyError, accuracy],
                     ↪ feed_dict={X: np.reshape(self.testData,
                     ↪ (self.testData.shape[0], 784)), y_target:
                     ↪ self.testTarget})
46              # Plot against currEpoch for Neural Network
47              xAxis.append(currEpoch)
```

```
            yTrainErr.append(errTrain)
            yTrainAcc.append(accTrain)
            yValidErr.append(errValid)
            yTestErr.append(errTest)
            yValidAcc.append(accValid)
            yTestAcc.append(accTest)
            currEpoch += 1
            logStdOut("e" + str(currEpoch))
        self.printPlotResults(numUpdate, errTrain, accTrain,
         ↪ errValid, errTest, accValid, accTest, xAxis,
         ↪ yTrainErr, yValidErr, yTestErr, yTrainAcc,
         ↪ yValidAcc, yTestAcc, hiddenImages)
```

```
def convertOneHot(targetValues):
    numClasses = np.max(targetValues) + 1
    return np.eye(numClasses)[targetValues]
```

```
def executeNeuralNetwork(questionTitle, numEpoch,
 ↪ learningRates, weightDecay, doDropout, doVisualization,
 ↪ hiddenLayers):
    startTime = datetime.datetime.now()
    with np.load("notMNIST.npz") as data:
        Data, Target = data ["images"], data["labels"]
        for learningRate in learningRates:
            for hiddenLayer in hiddenLayers:
                np.random.seed(521)
                randIndx = np.arange(len(Data))
                np.random.shuffle(randIndx)
                Data = Data[randIndx]/255.
```

```
                Target = Target[randIndx]
                trainData, trainTarget = Data[:15000],
                 ↪ Target[:15000]
                validData, validTarget = Data[15000:16000],
                 ↪ Target[15000:16000]
                # Target values are from 0 to 9
                testData, testTarget = Data[16000:],
                 ↪ Target[16000:]
                trainTarget = convertOneHot(trainTarget)
                validTarget = convertOneHot(validTarget)
                testTarget = convertOneHot(testTarget)
                tf.reset_default_graph()
                FullyConnectedNeuralNetwork(trainData,
                 ↪ trainTarget, validData, validTarget,
                 ↪ testData, testTarget, numEpoch,
                 ↪ learningRate, weightDecay, doDropout,
                 ↪ doVisualization, hiddenLayer,
                 ↪ questionTitle)
                logElapsedTime(questionTitle +
                 ↪ str(learningRate) + str(hiddenLayer))
```

```
# Global for logging
questionTitle = "" # Need to be global for logging to work
startTime = datetime.datetime.now()
figureCount = 1

def logStdOut(message):
    # Temporary print to std out
    sys.stdout = sys.__stdout__
```

```
    print message
    # Continue editing same file
    sys.stdout = open("result" + questionTitle + ".txt", "a")

def logElapsedTime(message):
    ''' Logs the elapsedTime with a given message '''
    global startTime
    endTime = datetime.datetime.now()
    elapsedTime = endTime - startTime
    hours, remainder = divmod(elapsedTime.seconds, 3600)
    minutes, seconds = divmod(remainder, 60)
    totalDays = elapsedTime.days
    timeStr = str(message) + ': Days: ' + str(totalDays) +  "
     ↪ hours: " + str(hours) + ' minutes: ' + str(minutes) +
     ↪ ' seconds: ' + str(seconds)
    logStdOut(timeStr)
    startTime = datetime.datetime.now()
```

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

```
    doVisualization = False
    hiddenLayers = [[1000]]
    logStdOut("Starting" + questionTitle)
    sys.stdout = open("result" + questionTitle + ".txt", "a")
    # append to file
    executeNeuralNetwork(questionTitle, numEpoch,
    learningRates, weightDecay, doDropout, doVisualization,
    hiddenLayers)
    logStdOut("Finished" + questionTitle)
    # '''

    '''
    # 2.3.1
    questionTitle = "2.3.1" # Number of hidden units
    numEpoch = 30 # Start to early stop at 5 for valid and 8
    for test
    learningRates = [0.1, 0.01, 0.001]
    learningRates = [0.001] # Works better when you run them
    one by one
    weightDecay = 3e-4
    doDropout = False
    doVisualization = False
    hiddenLayers = [[100], [500], [1000]]
    logStdOut("Starting" + questionTitle)
    #sys.stdout = open("result" + questionTitle + ".txt", "w")
    # write a new file from scratch
    sys.stdout = open("result" + questionTitle + ".txt", "a")
    # append to file
    executeNeuralNetwork(questionTitle, numEpoch,
    learningRates, weightDecay, doDropout, doVisualization,
    hiddenLayers)
```

```
    logStdOut("Finished" + questionTitle)
    # '''

    # 2.3.2
    questionTitle = "2.3.2" # Number of Layers
    numEpoch = 30 # Start to early stop at 5 for valid and 8 for test
    learningRates = [0.1, 0.01, 0.001]
    learningRates = [0.001] # Works better when you run them one by one
    weightDecay = 3e-4
    doDropout = False
    doVisualization = False
    hiddenLayers = [[500, 500]]
    logStdOut("Starting" + questionTitle)
    sys.stdout = open("result" + questionTitle + ".txt", "a") # append to file
    executeNeuralNetwork(questionTitle, numEpoch, learningRates, weightDecay, doDropout, doVisualization, hiddenLayers)
    logStdOut("Finished" + questionTitle)

    '''
    # 2.4 Regularization
    # Early Stops at about 12 for both valid at test
    questionTitle = "2.4.1" # Dropout
    numEpoch = 12
    learningRates = [0.1, 0.01, 0.001]
    weightDecay = 3e-4
    doDropout = True
```

```
    doVisualization = False
    hiddenLayers = [[1000]]
    logStdOut("Starting" + questionTitle)
    sys.stdout = open("result" + questionTitle + ".txt", "w")
↪   # write a new file from scratch
    executeNeuralNetwork(questionTitle, numEpoch,
↪   learningRates, weightDecay, doDropout, doVisualization,
↪   hiddenLayers)
    logStdOut("Finished" + questionTitle)
    # '''


    '''
    questionTitle = "2.4.2" # Visualization
    numEpoch = 12
    learningRates = [0.1]
    weightDecay = 3e-4
    doDropout = True
    doVisualization = True
    hiddenLayers = [1000]
    logStdOut("Starting" + questionTitle)
    sys.stdout = open("result" + questionTitle + ".txt", "w")
↪   # write a new file from scratch
    executeNeuralNetwork(questionTitle, numEpoch,
↪   learningRates, weightDecay, doDropout, doVisualization,
↪   hiddenLayers)
    logStdOut("Finished" + questionTitle)
    # '''
```

3.2 Entire Code: FuYuan Tee's version

3.2.1 Base Code for Q1

```
import tensorflow as tf
import numpy as np
import math

import time
import datetime

# Non-interactive plotting
import matplotlib.pyplot as plt
from IPython import display

# Interactive plotting
from plotly import tools
import plotly.plotly as py
import plotly.graph_objs as go
import plotly.offline as pyo
from plotly.offline import download_plotlyjs

# Configure environment
%config InlineBackend.figure_format = 'retina'
np.set_printoptions(precision=3)

# Global Variables
CURRENT_DIR =
 ↪ '/Users/christophertee/Dropbox/University/MASc/Courses/Winter
 ↪ 2017' + \
```

```
'/ECE521 (Inference Algorithms & Machine Learning)/Assignment
 ↪ 2'
LOG_DIR = '/Logs'

# Activate Plotly Offline for Jupyter
pyo.init_notebook_mode(connected=True)

# Load notMNIST dataset
'''
 Training Set: 3500 images
 Validation Set: 100 images
 Test Set: 145 images
 Images are 28 x 28 (normalised) pixels
'''
with np.load("notMNIST.npz") as data:
    Data, Target = data ["images"], data["labels"]

    # Subsetting data for classes 'C' (2) and 'J' (9)
    posClass = 2 # 'C'
    negClass = 9 # 'J'
    dataIndx = (Target==posClass) + (Target==negClass)

    Data = Data[dataIndx] / 255.
    Target = Target[dataIndx].reshape(-1, 1)

    # Converts target labels to 'C' (0) and 'J' (1)
    Target[Target==posClass] = 1
    Target[Target==negClass] = 0

    # Set random seed
```

```
54      np.random.seed(521)
55
56      # Generate and shuffle random index
57      randIndx = np.arange(len(Data))
58      np.random.shuffle(randIndx)
59
60      Data = Data[randIndx]
61      Target = Target[randIndx]
62
63      # Flatten arrays of dimension m x 28 x 28 into array of
        ↪ dimension m x 784
64      Data = Data.reshape(Data.shape[0], -1)
65
66      # Standardizing inputs of dataset
67      Data -= np.mean(Data, axis=0)
68      Data /= np.std(Data, axis=0)
69
70      # Partition data into training, validation and test
        ↪ datasets
71      Data, Target = Data[randIndx], Target[randIndx]
72      trainData, trainTarget = Data[:3500], Target[:3500]
73      validData, validTarget = Data[3500:3600],
        ↪ Target[3500:3600]
74      testData, testTarget = Data[3600:], Target[3600:]
75
76  MAX_ITER = 10000
77  def tuneLearningRate(etaList, model, section_dir,
    ↪ batchSize=500, lambda_=0.01, opt='Adam'):
78      # Returns the i-th batch of training data and targets
79      # Generates a new, reshuffled batch once all previous
        ↪ batches are fed
```

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

```
105         averageN =
             ↪ np.average(currentValidErr[-(np.arange(epochSize -
             ↪ 1) + 1)])
106         averageNlessOne =
             ↪ np.average(currentValidErr[-(np.arange(epochSize -
             ↪ 1) + epochSize)])
107         return averageN / averageNlessOne
108
109     # Returns True if the average epoch validation MSE is at
         ↪ least 99.9% of the previous epoch average.
110     # i.e. Returns True if the average learnings between epoch
         ↪ is less than +0.1%
111     # Otherwise, returns False
112     def shouldStopEarly(currentValidErr):
113         if currentValidErr.shape[0] < 2 * epochSize:
114             return False
115         return True if (ratioAverageEpochMSE(currentValidErr)
             ↪ > 0.999) else False
116
117
118     # Start of function
119     summaryList = []
120     randIdx = np.arange(trainData.shape[0])
121     epochSize = trainData.shape[0] / batchSize
122     randIdx = np.arange(trainData.shape[0])
123
124     assert section_dir
125     chapter_dir = '/Binary Loss/Section ' + section_dir + '/'
126     current_time = '{:%b%d
         ↪ %H_%M_%S}'.format(datetime.datetime.now())
```

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

```
                                                graph=sess.graph)

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

            # Creates blank training and validation MSE arrays
             ↪ for the Session
            currentTrainErr = np.array([])[:, np.newaxis]
            currentValidErr = np.array([])[:, np.newaxis]
            currentTestErr = np.array([])[:, np.newaxis]

            currentTrainAcc = np.array([])[:, np.newaxis]
            currentValidAcc = np.array([])[:, np.newaxis]
            currentTestAcc = np.array([])[:, np.newaxis]

            # Runs update
            currentIter = 0
            while currentIter < MAX_ITER:
                inputData, inputTarget =
                 ↪ getNextTrainingBatch(currentIter, randIdx)

                _, trainErr, trainAcc = sess.run([optimizer,
                 ↪ error, accuracy], feed_dict={X: inputData,
                 ↪ Y: inputTarget})
                validErr, validAcc = sess.run([error,
                 ↪ accuracy], feed_dict={X: validData, Y:
                 ↪ validTarget})
                testErr, testAcc = sess.run([error, accuracy],
                 ↪ feed_dict={X: testData, Y: testTarget})

```

```
172                 currentTrainErr = np.append(currentTrainErr,
                     ↪ trainErr)
173                 currentValidErr = np.append(currentValidErr,
                     ↪ validErr)
174                 currentTestErr = np.append(currentTestErr,
                     ↪ testErr)
175
176                 currentTrainAcc = np.append(currentTrainAcc,
                     ↪ trainAcc)
177                 currentValidAcc = np.append(currentValidAcc,
                     ↪ validAcc)
178                 currentTestAcc = np.append(currentTestAcc,
                     ↪ testAcc)
179
180                 # Update graph of training and validation MSE
                     ↪ arrays
181 #                 if (currentIter < 3) or (currentIter % 1000
     ↪ == 0):
182 #                     plotErrGraph([currentTrainErr,
     ↪ currentValidErr], eta)
183
184                 # At every epoch, check for early stopping
                     ↪ possibilty. If so, breaks from while loop
185                 if currentIter % epochSize == 0:
186                     if shouldStopEarly(currentValidErr):
187                         writer.close()
188                         break
189
190                 currentIter += 1
191
```

```
192                 if currentIter == MAX_ITER:
193                     writer.close()
194
195         # Save session results as dictionary and appends to
             ↪ MSEsummaryList
196         summaryList.append(
197             {
198                 'eta': eta,
199                 'B': batchSize,
200                 'lambda': lambda_,
201                 'optimizer': opt,
202                 'numIter': currentIter + 1,
203                 'epoch': float(currentIter + 1) / epochSize,
204                 'trainError': currentTrainErr,
205                 'validError': currentValidErr,
206                 'testError': currentTestErr,
207                 'trainAccuracy': currentTrainAcc,
208                 'validAccuracy': currentValidAcc,
209                 'testAccuracy': currentTestAcc
210             }
211         )
212
213         # Print stats when one eta run is done
214         print 'eta: %7.3f, numIter: %7d, validError: %.3f,
             ↪ testAcc: %.3f duration: %3.1fs' % \
215             (summaryList[-1]['eta'],
                 ↪ summaryList[-1]['numIter'],
                 ↪ summaryList[-1]['validError'][-1], \
216
                   ↪ np.mean(summaryList[-1]['testAccuracy'][-epochSize:]),
                   ↪ 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,
                       ↪ None], name='input')
9                  Y = tf.placeholder(tf.float32, shape=[None,
                       ↪ None], name='target')
10
11             # Model parameters
12             with tf.name_scope('parameters'):
13                 W = tf.get_variable('weights', shape=[784, 1],
                       ↪ initializer=tf.truncated_normal_initializer(stddev=0.
14                 b = tf.get_variable('biases', shape=[1, 1],
                       ↪ initializer=tf.constant_initializer(0.0))
15
16             with tf.device('/cpu:0'):
17                 # Predicted target
18                 with tf.name_scope('prediction'):
19                     Z = tf.add(tf.matmul(X, W), b, 'logits')
20                     Yhat = tf.sigmoid(Z, name='pred')
```

```
21
22                  # Metrics
23                  with tf.name_scope('metrics'):
24                      with tf.name_scope('error'):
25                          error =
                             ↪ tf.add(tf.reduce_mean(tf.nn.sigmoid_cross_ent
                             ↪ labels=Y, name='elem_x_entropy'),
                             ↪ \
26
                                                                 ↪ name='total_x_e
                                                                 ↪ \
27                                       tf.multiply(lambda_ /
                                          ↪ 2,
                                          ↪ tf.matmul(tf.transpose(W),
                                          ↪ W),
                                          ↪ name='l2_loss'), \
28                                       name='total_loss')
29                      with tf.name_scope('threshold'):
30                          YhatThres =
                             ↪ tf.cast(tf.greater_equal(Yhat,
                             ↪ 0.5, name='pred_thres'),
                             ↪ tf.float32)
31                      with tf.name_scope('accuracy'):
32                          accuracy =
                             ↪ tf.truediv(tf.reduce_sum(tf.cast(tf.equal(Yha
                             ↪ Y), tf.int32),
                             ↪ name='total_matches'), \
33                                                   tf.shape(X)[0],
                                                      ↪ \
34                                                   name='accuracy')
```

```
35
36          # Optimizer
37          with tf.device('/cpu:0'):
38              if opt == 'GD':
39                  optimizer =
                     ↪ tf.train.GradientDescentOptimizer(eta).minimize(error
40              elif opt == 'Adam':
41                  optimizer =
                     ↪ tf.train.AdamOptimizer(eta).minimize(error)
42
43      return W, b, X, Y, YhatThres, error, accuracy, optimizer
44
45  etaList = [0.001, 0.01, 0.1, 1, 10]
46  summary1_1 = tuneLearningRate(etaList, model='logistic',
     ↪ 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',
    ↪ 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             with tf.name_scope('placeholders'):
7                 X = tf.placeholder(tf.float32, shape=[None,
                      ↪ None], name='Input')
8                 Y = tf.placeholder(tf.float32, shape=[None,
                      ↪ None], name='Target')
9
10            # Model parameters
11            with tf.name_scope('parameters'):
12                W = tf.get_variable('weights', shape=[784, 1],
                      ↪ initializer=tf.truncated_normal_initializer(stddev=0.
13                b = tf.get_variable('biases', shape=[1, 1],
                      ↪ initializer=tf.constant_initializer(0.0))
14
15            with tf.device('/cpu:0'):
16                # Predicted target
17                with tf.name_scope('prediction'):
18                    Yhat = tf.add(tf.matmul(X, W), b, 'pred')
19
20                # Metrics
21                with tf.name_scope('metrics'):
22                    with tf.name_scope('error'):
23                        error =
                              ↪ tf.add(tf.reduce_mean(tf.nn.l2_loss(tf.subtra
                              ↪ Y), name='elem_l2_loss'), \
24
                                                          ↪ name='total_l2_
                                                          ↪ \
25                                                    tf.multiply(lambda_ /
                                                      ↪ 2,
                                                      ↪ tf.matmul(tf.transpose(W),
                                                      ↪ W),
                                                      ↪ name='l2_loss'), \
```

```
                                        name='total_loss')
                    with tf.name_scope('threshold'):
                        YhatThres =
                         ↪ tf.cast(tf.greater_equal(Yhat,
                         ↪ 0.5, name='pred_thres'),
                         ↪ tf.float32)
                    with tf.name_scope('accuracy'):
                        accuracy =
                         ↪ tf.truediv(tf.reduce_sum(tf.cast(tf.equal(Yha
                         ↪ Y), tf.int32),
                         ↪ name='total_matches'), \
                                                tf.shape(X)[0],
                                                 ↪ \
                                                name='accuracy')

        # Optimizer
        with tf.device('/cpu:0'):
            optimizer =
             ↪ tf.train.AdamOptimizer(eta).minimize(error)

    return W, b, X, Y, YhatThres, error, accuracy, optimizer

etaList = [1e-4, 1e-3, 1e-2, 1e-1, 1, 10]
print 'Linear Regression:'
summary1_3_linear = tuneLearningRate(etaList, model='linear',
 ↪ section_dir='1.3Lin', lambda_=0.0)
print '\nLogistic Regression:'
summary1_3_logistic = tuneLearningRate(etaList,
 ↪ 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) *
        ↪ np.log(1 - Yhat)
53     squaredErr = (Yhat - Ydummy)**2
54 
55     # Generate plot traces
56     xEntropyTrace = go.Scatter(
57         x = Yhat,
58         y = xEntropy,
59         name = 'Cross-Entropy'
60     )
61     squareErrTrace = go.Scatter(
62         x = Yhat,
63         y = squaredErr,
64         name = 'Squared-Error'
65     )
66 
67     data = go.Data([xEntropyTrace, squareErrTrace])
68 
69     # Generate figure layout
70     layout = go.Layout(
71         title = '$\\text{Comparison of Cross Entropy and
            ↪ Squared Error Losses on Dummy Variable } y = 0$',
72         xaxis = {'title': '$\\hat{y}$'},
73         yaxis = {'title': 'Loss'}
```

```
    )

    # Creates and plots figure
    figure = go.Figure(data=data, layout=layout)
    return py.iplot(figure,
     ↪ filename='A2Q1.1.3_LossComparison')

fig1_3 = generateDummyLosses()
fig1_3
```

3.2.5 Q1.2.3: Multi-class classification

```
# TensorFlow Softmax Modeldef buildSoftmaxGraph(eta, lambda_):
    with tf.device('/cpu:0'):
        with tf.variable_scope('softmax'):
            # Model inputs
            with tf.name_scope('placeholders'):
                X = tf.placeholder(tf.float32, shape=[None,
                 ↪ None], name='input')
                Y = tf.placeholder(tf.float32, shape=[None,
                 ↪ None], name='target')

            # Model parameters
            with tf.name_scope('parameters'):
                W = tf.get_variable('weights', shape=[784,
                 ↪ 10],
                 ↪ initializer=tf.truncated_normal_initializer(stddev=0.
                b = tf.get_variable('biases', shape=[1, 10],
                 ↪ 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 =
                         ↪ tf.reduce_mean(tf.nn.softmax_cross_entropy_with_l
                         ↪ \
25
26
27
                                                    ↪ name='total_x_entropy'
28                  with tf.name_scope('l2_loss'):
29                      l2_loss = tf.multiply(lambda_ / 2, \
30
                                                ↪ tf.reduce_sum(tf.matmul(tf.
                                                ↪ W)),
                                                ↪ name='l2_loss')
31                  with tf.name_scope('total_loss'):
32                      error = tf.add(x_entropy, l2_loss,
                         ↪ name='total_loss')
```

```
33              with tf.name_scope('threshold'):
34                  YhatThres = tf.cast(tf.argmax(Yhat, axis=1,
                     ↪  name='pred_threshold'), tf.float32)
35              with tf.name_scope('accuracy'):
36                  Ycollapsed = tf.cast(tf.argmax(Y, axis=1),
                     ↪  tf.float32, name='target_reverse_one_hot')
37                  accuracy =
                     ↪  tf.truediv(tf.reduce_sum(tf.cast(tf.equal(YhatThres,
                     ↪  Ycollapsed) \
38                                                                  ,
                                                                     ↪  tf.int32)
                                                                     ↪  \
39
                                                                  ↪  name='total_match
                                                                  ↪  \
40                                          tf.shape(X)[0], \
41                                          name='accuracy')
42
43      # Optimizer
44      with tf.device('/cpu:0'):
45          optimizer =
             ↪  tf.train.AdamOptimizer(eta).minimize(error)
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
```

```
 Images are 28 x 28 (normalised) pixels
'''
with np.load("notMNIST.npz") as data:
    Data, Target = data ["images"], data["labels"]

    # Set random seed
    np.random.seed(521)

    # Generate and shuffle random index
    randIndx = np.arange(len(Data))
    np.random.shuffle(randIndx)

    Data = Data[randIndx]/255.
    Target = Target[randIndx]

    # Generates one-hot version of target
    oneHot = np.zeros((Target.shape[0], 10))
    oneHot[np.arange(Target.shape[0]), Target] = 1
    Target = oneHot

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

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

    # Partition data into training, validation and test
     ↪ datasets
```

```
    trainData, trainTarget = Data[:15000], Target[:15000]
    validData, validTarget = Data[15000:16000],
     ↪ Target[15000:16000]
    testData, testTarget = Data[16000:], Target[16000:]
```

3.2.6 Base Code for Q2

```
import tensorflow as tf
import numpy as np
import math

import time
import datetime

# Non-interactive plotting
import matplotlib.pyplot as plt
from IPython import display

# Interactive plotting
from plotly import tools
import plotly.plotly as py
import plotly.graph_objs as go
import plotly.offline as pyo
from plotly.offline import download_plotlyjs

# Configure environment
%config InlineBackend.figure_format = 'retina'
np.set_printoptions(precision=3)

```

```
# Global Variables
CURRENT_DIR =
 ↪ '/Users/christophertee/Dropbox/University/MASc/Courses/Winter
 ↪ 2017' + \
'/ECE521 (Inference Algorithms & Machine Learning)/Assignment
 ↪ 2'
LOG_DIR = '/Logs'

# Activate Plotly Offline for Jupyter
pyo.init_notebook_mode(connected=True)

# Load notMNIST dataset
'''
 Training Set: 15,000 images
 Validation Set: 1,000 images
 Test Set: 2,724 images
 Images are 28 x 28 (normalised) pixels
'''
with np.load("notMNIST.npz") as data:
    Data, Target = data ["images"], data["labels"]

    # Set random seed
    np.random.seed(521)

    # Generate and shuffle random index
    randIndx = np.arange(len(Data))
    np.random.shuffle(randIndx)

    Data = Data[randIndx]/255.
    Target = Target[randIndx]
```

```
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
        ↪ dimension m x 784
58      Data = np.reshape(Data, (Data.shape[0], 784))
59
60      # Standardizing inputs of dataset
61      Data -= np.mean(Data, axis=0)
62      Data /= np.std(Data, axis=0)
63
64      # Partition data into training, validation and test
        ↪ datasets
65      trainData, trainTarget = Data[:15000], Target[:15000]
66      validData, validTarget = Data[15000:16000],
        ↪ Target[15000:16000]
67      testData, testTarget = Data[16000:], Target[16000:]
68
69  # TensorFlow Fully-Connected Neural Network
70  def build_FCNN(eta, lambda_, numHidLayers, layerUnits,
     ↪ dropout_rate):
71      NUM_FEATURES = 784
72      NUM_TARGET_CATEGORIES = 10
73      '''
74      - Builds a fully-connected neural network with
     ↪ 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;
    ↪ len(layerUnits) == numHidLayer
82    '''
83
84    def connect_layers(h, numNextUnit, name='input_to_layer'):
85        '''
86        - Creates weights and biases to connect previous layer
    ↪ to next layer
87        - Returns the weighted sum of inputs for to the next
    ↪ layer
88        - Initializes weights using Xavier initialization.
89
90        Inputs:
91            hIn: Incoming activation tensor from previous
    ↪ layer
92            numOut: Number of hidden units of next layer
93
94        Outputs:
95            z: Weighted sum of inputs into the next layer
96        '''
97        numPrevUnit = h.get_shape()[1]
98
99        # Xavier initialization for weights
100       xavier = tf.sqrt(3 / tf.cast(numPrevUnit +
           ↪ numNextUnit, tf.float32), name='xavier')
```

```
101
102         W = tf.get_variable('weights', shape=[numPrevUnit,
             ↪ numNextUnit], \
103
                                 ↪ initializer=tf.truncated_normal_initializ
                                 ↪ seed=1))
104         b = tf.get_variable('biases', shape=[1, numNextUnit],
             ↪ initializer=tf.constant_initializer(0.0))
105
106         tf.add_to_collection(tf.GraphKeys.WEIGHTS, W)
107         tf.add_to_collection(tf.GraphKeys.BIASES, b)
108
109         return tf.add(tf.matmul(h, W), b, name=name)
110
111
112     def build_layers(numHidLayers, layerUnits, dropout_rate):
113         '''
114         - Builds hidden layers and output layer
115
116         Inputs:
117             numHidLayers: Number of hidden layers
118             layerUnits: List of number of units in each layer;
    ↪ len(layerUnits) == numHidLayer
119
120         Output:
121             zOut: Logits to output layer
122         '''
123         ### Builds hidden layers
124         ZList = [] # Stores the inputs to the i-th layer
125         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
                 ↪ scope:
130                 # Calculates input to the i-th hidden layer
131                 # Uses placeholder tensor X instead of Z_(i-1)
                     ↪ in connectLayers for the first hidden
                     ↪ layer
132                 if i == 0:
133                     ZList.append(connect_layers(X,
                         ↪ layerUnits[i]))
134                 else:
135                     ZList.append(connect_layers(hList[-1],
                         ↪ layerUnits[i]))
136
137                 # Calculates ReLU activation of each layer
138
                     ↪ hList.append(tf.nn.dropout(tf.nn.relu(ZList[i],
                     ↪ name='hidden_activation'),
                     ↪ keep_prob=1-dropout_rate, \
139
                                                 ↪ name='dropped_activations')
140
141         # Adds activation variables into GraphKey.ACTIVATIONS
             ↪ collection
142         for h in hList:
143             tf.add_to_collection(tf.GraphKeys.ACTIVATIONS, h)
144
145         ### Builds output layer
```

```
        with tf.variable_scope('output_layer'):
            ZList.append(connect_layers(hList[-1],
             ↪ NUM_TARGET_CATEGORIES, name='logits'))

        return ZList[numHidLayers]


    ########################################
    #           Function begins            #
    ########################################
    mode = '/gpu:0'
    tf.set_random_seed(521)

    assert len(layerUnits) == numHidLayers

    with tf.device('/cpu:0'):
        with tf.name_scope('placeholders'):
            X = tf.placeholder(tf.float32, shape=[None,
             ↪ NUM_FEATURES], name='input_layer')
            Y = tf.placeholder(tf.float32, shape=[None,
             ↪ NUM_TARGET_CATEGORIES], name='target')

    with tf.device(mode):
        logits = build_layers(numHidLayers, layerUnits,
         ↪ dropout_rate) # Logits to output layer

        # Calculates prediction
        with tf.name_scope('prediction'):
            Yhat = tf.nn.softmax(logits, name='pred')
            YhatThres = tf.cast(tf.argmax(Yhat, axis=1,
             ↪ 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 =
                        ↪ tf.reduce_mean(tf.nn.softmax_cross_entropy_with_l
                        ↪ \
181
182
183
                                                ↪ name='total_x_entropy')
184                 # Calculates l2 loss
185                 with tf.name_scope('l2_loss'):
186                     weights =
                        ↪ tf.get_collection_ref(tf.GraphKeys.WEIGHTS)
187                     biases =
                        ↪ tf.get_collection_ref(tf.GraphKeys.BIASES)
188                     l2_loss = tf.multiply(lambda_ / 2, \
189
                                                ↪ tf.add_n([tf.nn.l2_loss(W)
                                                ↪ for W in
                                                ↪ weights]),
                                                ↪ name='total_l2_loss')
```

```
190                 # Calculates total loss
191                 with tf.name_scope('total_loss'):
192                     error = tf.add(xent, l2_loss,
                        ↪  name='total_loss')
193             # Calculates accuracy
194             with tf.name_scope('accuracy'):
195                 Ycollapsed = tf.cast(tf.argmax(Y, axis=1),
                    ↪  tf.float32, name='target_reverse_one_hot')
196                 accuracy =
                    ↪  tf.truediv(tf.reduce_sum(tf.cast(tf.equal(YhatThres,
                    ↪  Ycollapsed) \
197                                                                 ,
                                                                    ↪  tf.int32)
                                                                    ↪  \
198
                                                                ↪  name='total_match
                                                                ↪  \
199                                         tf.shape(X)[0], \
200                                         name='accuracy')
201     with tf.device(mode):
202         # Optimizer
203         with tf.variable_scope('Adam'):
204             optimizer =
                ↪  tf.train.AdamOptimizer(eta).minimize(error)
205
206     return X, Y, YhatThres, weights, error, accuracy,
        ↪  optimizer
207
208 # Define training function
209 def train_NN(etaList, numHidLayersList, layerUnitsList,
    ↪  sectionDir, batchSize=500, lambda_=3e-4, dropout_rate=0):
```

```
210     MAX_ITER = 3000
211     '''
212     - Returns the i-th batch of training data and targets
213     - Generates a new, reshuffled batch once all previous
    ↪  batches are fed
214     '''
215     def get_next_training_batch(currentIter, randIdx):
216         currentBatchNum = currentIter % (trainData.shape[0] /
            ↪  batchSize)
217         if currentBatchNum == 0:
218             np.random.shuffle(randIdx)
219         lowerBoundIdx = currentBatchNum * batchSize
220         upperBoundIdx = (currentBatchNum + 1) * batchSize
221         return
            ↪  trainData[randIdx[lowerBoundIdx:upperBoundIdx]],
            ↪  trainTarget[randIdx[lowerBoundIdx:upperBoundIdx]]
222
223     # Generate updated plots for training and validation MSE
224     def plot_err_graph(errList, param):
225         label = '$\eta$ = ' + str(param)
226         label_classification = ['train.', 'valid.']
227
228         display.clear_output(wait=True)
229         plt.figure(figsize=(8,5), dpi=200)
230
231         for i, err in enumerate(errList):
232             plt.plot(range(len(err)), err, '-', label=label+'
                ↪  '+label_classification[i])
233
234         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
        ↪ and the (n-1)-th average epoch MSE
239     def average_epoch_err_ratio(validErr):
240         averageN = np.mean(validErr[-epochSize:])
241         averageNlessOne =
            ↪ np.mean(validErr[-epochSize*2:-epochSize])
242         return averageN / averageNlessOne
243 
244     # Returns True if the average epoch validation MSE is at
        ↪ least 99.9% of the previous epoch average.
245     # i.e. Returns True if the average learnings between epoch
        ↪ is less than +0.1%
246     # Otherwise, returns False
247     def should_stop_early(validErr):
248         if validErr.shape[0] < 2 * epochSize:
249             return False
250         return True if (average_epoch_err_ratio(validErr) > 1)
            ↪ else False
251 
252     # Start of function
253     summaryList = []
254     epochSize = trainData.shape[0] / batchSize
255 
256     assert sectionDir
257 
258     for eta in etaList:
259         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 ' +
                 ↪ sectionDir + '/Eta_' + str(eta) + '/'
274             currentTime = '{:%b%d
                 ↪ %H_%M_%S}'.format(datetime.datetime.now())
275             writeDir = CURRENT_DIR + LOG_DIR + chapterDir +
                 ↪ currentTime
276
277             # Build new graph
278             tf.reset_default_graph() # Reset graph to prevent
                 ↪ duplication of ops and variables
279             X, Y, YhatThres, W, error, accuracy, optimizer =
                 ↪ build_FCNN(eta, lambda_, numHidLayers,
                 ↪ layerUnits, dropout_rate)
280             # Create saver
281         #     saver = tf.train.Saver()
282
283             # Begin session
```

```
284             with tf.Session(config=tf.ConfigProto(log_device_placement=Fal
                ↪ as sess:
285                 # Log starting time
286                 startTime = time.time()
287
288                 # Create summary writer
289                 writer = tf.summary.FileWriter(writeDir,
                    ↪ graph=sess.graph)
290
291                 # Initialise all TensorFlow variables
292                 tf.global_variables_initializer().run()
293
294                 # Create blank training, validation and test
                    ↪ error arrays for the Session
295                 trainErr = np.array([])[:, np.newaxis]
296                 validErr = np.array([])[:, np.newaxis]
297                 testErr = np.array([])[:, np.newaxis]
298
299                 # Create blank training, validation and test
                    ↪ accuracies arrays for the Session
300                 trainAcc = np.array([])[:, np.newaxis]
301                 validAcc = np.array([])[:, np.newaxis]
302                 testAcc = np.array([])[:, np.newaxis]
303
304                 # Create blank arrays to store weights for the
                    ↪ Session
305                 weights_l0 = np.array([])[:, np.newaxis]
306
307                 # 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_l0 = sess.run(W)[0][:,:,
                                ↪ np.newaxis]
314                         else:
315                             weights_l0 = np.append(weights_l0,
                                ↪ 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})
```

```
                # Append run values
                trainErr = np.append(trainErr,
                 ↪ currTrainErr)
                validErr = np.append(validErr,
                 ↪ currValidErr)
                testErr = np.append(testErr, currTestErr)

                trainAcc = np.append(trainAcc,
                 ↪ currTrainAcc)
                validAcc = np.append(validAcc,
                 ↪ currValidAcc)
                testAcc = np.append(testAcc, currTestAcc)

                # Perform optimization routine
                sess.run(optimizer, feed_dict={X:
                 ↪ inputData, Y: inputTarget})

                # Provide info on training progress at the
                 ↪ end of every 10th epoch
                if currentIter == 0 or currentIter %
                 ↪ (epochSize * 10) == 0:
                    print 'epoch: %5.0f, currTrainErr:
                     ↪ %10.2f, currValidErr: %10.2f' % \
                        ((currentIter + 1) / epochSize,
                         ↪ currTrainErr, currValidErr)
#                   plot_err_graph([trainErr, validErr],
 ↪ eta)

                # At the end of each epoch, check for
                 ↪ early stopping possibilty. If so,
                 ↪ break from while loop.
```

```
                    if (currentIter + 1) % epochSize == 0:
                        if should_stop_early(validErr):
                            writer.close()
                            print 'Exited from early
                             ↪ stopping\n'
                            break

                    # At every 10 epochs, creates checkpoint
                     ↪ for session
#                     if currentIter % epochSize == 10:
#                         saver.save(sess, writeDir,
 ↪ global_step=currentIter)

                    currentIter += 1

                    if currentIter == MAX_ITER:
                        print 'MAX_ITER reached\n'
                        writer.close()

            # Save session results as dictionary and appends
             ↪ to summaryList
            summaryList.append(
                {
                    'eta': eta,
                    'B': batchSize,
                    'lambda': lambda_,
                    'hidden_layers': numHidLayers,
                    'hidden_units': layerUnits,
                    'numIter': currentIter + 1 - epochSize,
                    'epoch': float(currentIter + 1) /
                     ↪ epochSize - 1,
```

```
370                     'trainError':
                         ↪ trainErr[::epochSize].copy()[::],
371                     'validError':
                         ↪ validErr[::epochSize].copy()[::],
372                     'testError':
                         ↪ testErr[::epochSize].copy()[::],
373                     'trainAccuracy':
                         ↪ trainAcc[::epochSize].copy()[::],
374                     'validAccuracy':
                         ↪ validAcc[::epochSize].copy()[::],
375                     'testAccuracy':
                         ↪ testAcc[::epochSize].copy()[::],
376                     'weights_layer_0': weights_l0
377                 }
378             )
379
380             # Print stats when a run is done
381             print 'epochs: %.1f, currentIter: %d' %
                 ↪ (summaryList[-1]['epoch'],
                 ↪ summaryList[-1]['numIter'])
382             print 'trainError: %.3f, validError: %.3f,
                 ↪ testError: %.3f' % \
383                     (summaryList[-1]['trainError'][-1],
                         ↪ summaryList[-1]['validError'][-1],
                         ↪ summaryList[-1]['testError'][-1])
384             print 'Max trainAccuracy: %.3f, Max validAccuracy:
                 ↪ %.3f, Max testAccuracy: %.3f' % \
385                     (np.max(trainAcc), \
386                      np.max(validAcc), \
387                      np.max(testAcc))
```

```
388             print 'runTime: %3.1fs\n\n' % (time.time() -
                ↪ startTime)
389
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],
  ↪ numHidLayersList=[1], layerUnitsList=[[1000]],
  ↪ 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,
  ↪ 1, 1], layerUnitsList=[[100], [500], [1000]],
  ↪ 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], layerUnitsList=[[1000], [500, 500]],
  ↪ sectionDir='2.3.2')
```

3.2.10 Q2.4.1: Dropout

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

3.2.11 Q2.4.2: Visualization

```
# Create visualization method
'''
Creates four plots at 25%, 50%, 75% and 100% of training
 ↪ epochs.
Each plot consists of 40x25 subplots of 28x28 heatmaps to
 ↪ represent each of the 1000x28x28 weights.
'''
def visualise_weights(weights, hasDropout):
    # Mark start time
    startTime = time.time()

    # Calculate total number of epochs ran
    numEpochs = weights.shape[2]

    # Loop through 25%, 50%, 75%, 100% of runs
    for i in range(4):
        fig = plt.figure(figsize=(16,20))
        # Loop through all 1000 weights in the hidden layer
        for j in range(weights.shape[1]):
            # Plots on 40x25 subplot
            plt.subplot(40, 25, j + 1)
```

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

3.2.12 Q2.5.1: Random search

```
def run_exhaustive_search():
    '''
    Performs exhaustive search on hyperparameters by sampling from a random uniform distribution.
    Seed number used is a concatenation of the last two digits of both our student numbers. ('37' + '93')

    Selects random values for:
        ln(eta):             [-7.5, -4.5]
        hidden layers:       [1, 5]
        hidden layer units: [100, 500]
        ln(lambda):          [-9, -6]
        dropout:             'yes' or 'no'
    '''
    # Set random seed
    np.random.seed(3793)

    # Create placeholders to record randomly generated hyperparameters
    etaList = []
    layersList = []
    unitsList = []
    lambdaList = []
    dropoutList = []
    summary2_5_1 = []

    # Assign randomly generated values for model hyperparameters
```

```
    for i in range(5):
        # Randomly sample values for hyperparameters
        etaList.append(np.exp(np.random.uniform(-7.5, -4.5,
         ↪ size=1)))
        layersList.append(np.random.randint(1, 6, size=1))
        unitsList.append(np.random.randint(100, 501,
         ↪ size=layersList[i]))
        lambdaList.append(np.exp(np.random.uniform(-9, -6,
         ↪ size=1)).astype(np.float32))
        dropoutList.append(np.random.randint(0, 2))

    # Train each model
    for i in range(5):
        # Perform training run
        summary2_5_1.append(train_NN(etaList[i],
         ↪ layersList[i], [unitsList[i]], \
                                     ↪ dropout_rate=0.5*dropoutList[i],
                                     ↪ lambda_=lambdaList[i],
                                     ↪ sectionDir='2.5.1')[0])

    return summary2_5_1

summary2_5_1 = run_exhaustive_search()
```

3.2.13 Q2.5.2: Exchange ideas among the group

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