ECE521 - Assignment 2 - Logistic Regression and Neural Networks

K. L. Barry Fung - 999871125
Wen Tao Zhao - 999721127

Due February 27th, 2017

1 Logistic regression

1.1 Binary cross-entropy loss

Problem 1.1.1. *textbf Learning : Write a TensorFlow script that implements logistic regression and the crossentropy loss (and you want to use the tf.nn.sigmoid_cross_entropy_with_logits utility function to avoid numerical underflow). Set the weight-decay coefficient* $\lambda = 0.01$. *Train the logistic regression model using SGD and a mini-batch size* $B = 500$ *on the two-class notMNIST dataset. Plot the best training and testing curves for both cross-entropy loss and classification accuracy vs. the number of updates after tuning the learning rate. Report the best test classification accuracy obtained from the logistic regression model.*

Solution. The best test classification accuracy obtained from the logistic regression model was approximately 98.5%. This can be seen in Figures 1-2 below.

Figure 1: Logistic regression test accuracy

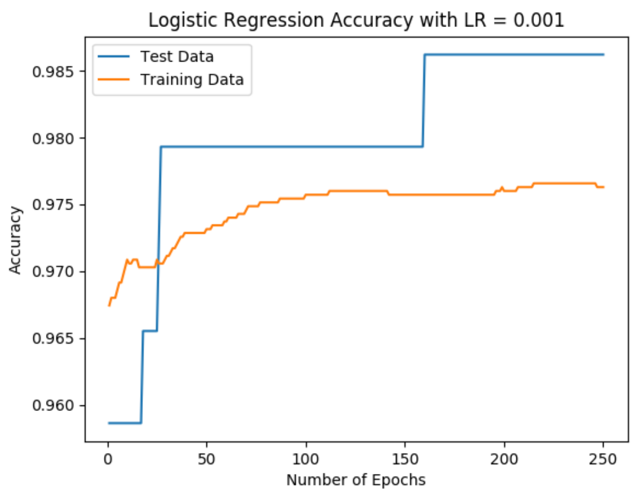
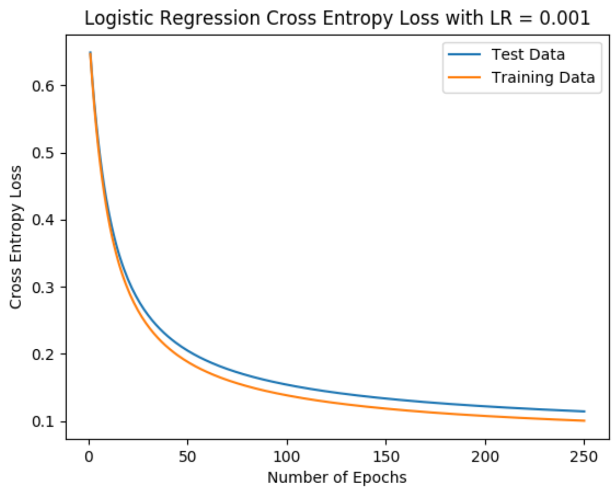


Figure 2: Logistic regression cross entropy loss



■

Problem 1.1.2. ***Beyond plain SGD:*** *With the same weight-decay coefficient = 0.01 and mini-batch size B = 500, train the logistic regression model with the Adam optimizer (use the tf.train.AdamOptimizer function). Plot the best training and testing curves for both the cross-entropy loss and the classification accuracy after tuning the learning rate. Compare the two sets of learning plots: those from SGD and those from using the Adam-Optimizer curve. Comment on their differences, and explain how the Adam optimizer may help learning from the notMNIST dataset. (For the rest of the assignment, you should use the Adam optimizer in all numerical experiments.)*

Solution. The Adam optimizer produced better results than the standard SGD optimizer in a lower number of epochs. This is due to the implementation in TensorFlow using moving averages of the parameters. This allows the Adam optimizer to effectively use a larger step size. However, the training step took slightly longer due to the additional computational steps. This was not an issue in our case but may be notable in an environment with more data.

Figure 3: Logistic regression test accuracy with Adam

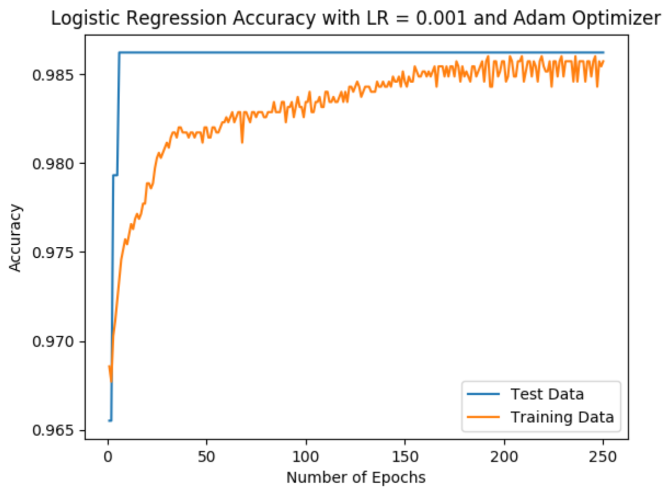
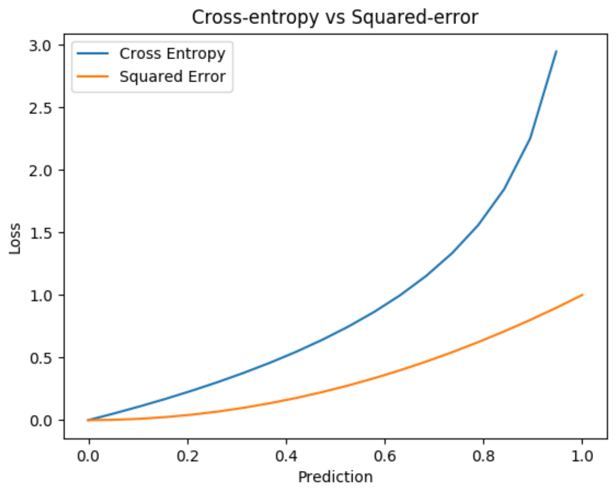


Figure 4: Logistic regression cross entropy loss with Adam



■

Problem 1.1.3. ***Comparison with linear regression*** *Set the weight decay coefficient to zero. Write a TensorFlow script to find the optimal linear regression weights on the two-class notMNIST dataset using the normal equation of the least squares formula. Compare the train, validation and test classification of the least squares solution to the optimal logistic regression learnt without weight decay. What do you observe about the effect of the cross-entropy loss on the classification performance? To help explain your observation, you may wish to plot the cross-entropy loss vs squared-error loss as a function of the prediction* $\hat{y}$ *within the interval [0, 1] and a dummy target* $y = 0$ *in 1-D.*

Solution. The logistic regression classifier performed better than the linear regression classifier. This is mainly due to the fact that cross-entropy loss is less affected by outliers than the standard least squares error. This is illustrated in the following graph of cross entropy and squared error loss.

Figure 5: linear regression test accuracy

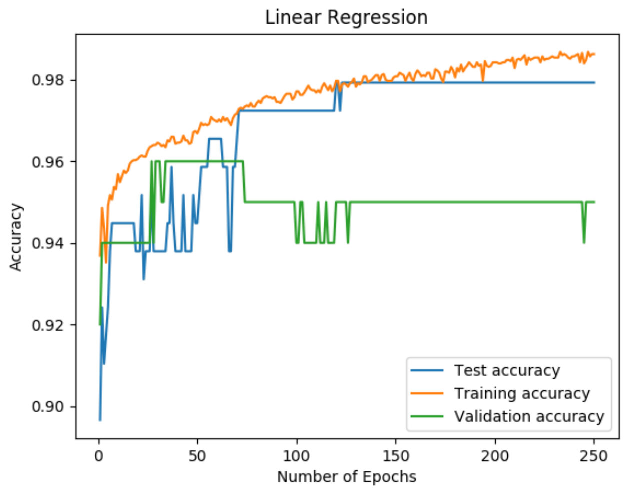
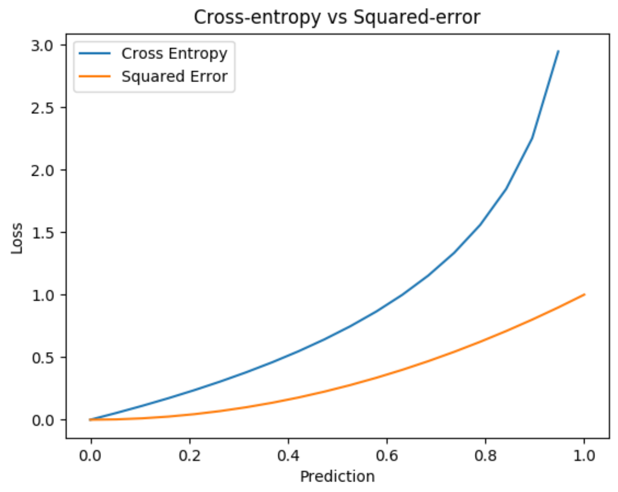


Figure 6: Error graphs



■

Problem 1.1.4. ***Maximum likelihood estimation:*** *In lecture we derived, from an i.i.d. data set, the MLE parameters of a Gaussian whose mean is parameterized by a linear function using weights W and b. It turned out that the MLE method was equivalent to minimizing the squared L2 loss from linear regression. Similarly, the cross-entropy loss can be derived from the MLE principle by assuming a Bernoulli distribution for the likelihood of the training label $y \in \{0, 1\}$, i.e.$P(y = 1|\boldsymbol{x}, W) = 1 - P(y = 0|\boldsymbol{x}, W) = \hat{y}(\boldsymbol{x})$. We can write the Bernoulli distributio parameterized by W more concisely as $P(y|\boldsymbol{x}) = \hat{y}(\boldsymbol{x})^y(1 - \hat{y}(\boldsymbol{x}))^{(1-y)}$ Show that minimizing the cross-entropy loss is equivalent to maximizing the log-likelihood of the training data under the Bernoulli assumption.*

Solution. To show this relation, we demonstrate that maximizing the log-likelihood will result in the minimization of the cross entropy, as shown below. The maximum of the log-likelihood is:

$$\begin{aligned}\max\{\log(P(y|x,w))\} &= \max\Big(\log\big(\hat{y}(x)^y(1-\hat{y}(x))^{1-y}\big)\Big)\\ &= \max\big(y\log(\hat{y}(x)) + (1-y)\log(1-\hat{y})\big)\\ &= \min\big(-y\log(\hat{y}(x)) - (1-y)\log(1-\hat{y})\big)\end{aligned}$$

This last statement is the expression of minimizing the cross-entropy loss, as required. ■

1.2 Multi-class classification

Problem 1.2.1. *Suppose a classifier operates by assigning each input, x, to the class with the largest a posteriori probability. Show that this classifier represents a minimum-loss system if the penalty for misclassification is equal for each class.*

Solution. The classifier aims to minimize the expected error of each classification. As the expected error is the sum of the probability of each misclassification multiplied by the associated penalty, equalizing the penalty terms would in effect reduce the expected error to the sum of the probability of misclassification. It then follows to select the class with the largest a posteriori probability in order to minimize the expected error ■

Problem 1.2.2. *Consider a more complex loss matrix, $\boldsymbol{L}$, in which L_{jk} represents the penalty for misclassification to class j when the pattern in fact belongs to class k. Under what condition should classification of x to class C_j be made if the total loss is to be minimized?*

Solution. Classification of x to class C_j should only be made if the expected error is minimized. That is to say, only the sum of all loss terms L_{jk} multiplied by the probability of x being in class k is minimized. This sum is given as:

$$\sum_k L_{kj}P(C_k, x \in R_j)$$

■

Problem 1.2.3. *Write a TensorFlow script that implements logistic regression using softmax output and with multi-class cross-entropy loss (you will want to use the tf.nn.softmax_cross _entropy_with_logits utility function, to avoid numerical underflow). Regularize the weights with = 0.01. Using a mini-batch size of B = 500 and a learning rate that you tune, train the logistic regression model on the full notMNIST dataset with 10 classes. For the best value of learning rate, plot the training and testing curves for both cross-entropy loss and classification accuracy vs. the number of updates. Report the best test classification accuracy obtained from the logistic regression model. Is the training and testing classification performance better or worse than the binary-class problem in question 1.1? Explain intuitively the difference.*

Solution. The best training accuracy from the logistic regression model is about 98.5%. The testing and training classification for the multi-class classifier is worse than the binary-class problem in question 1.1. This is to be expected as classifying ten different digits is intuitively harder than classifying only two. The script for performing this classification is included with the report.

Figure 7: Multiclass Classifier accuracy

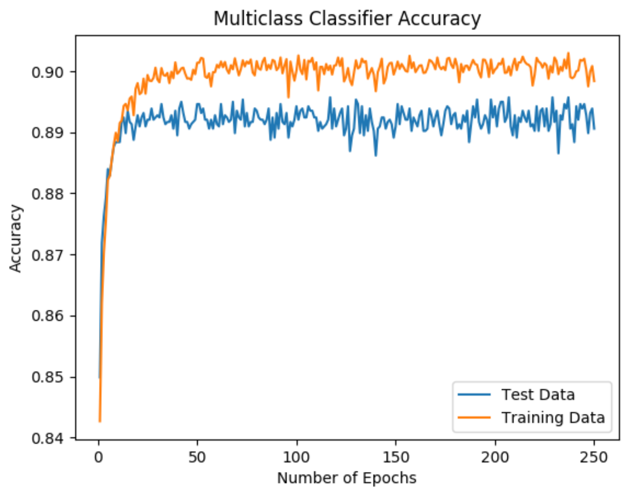
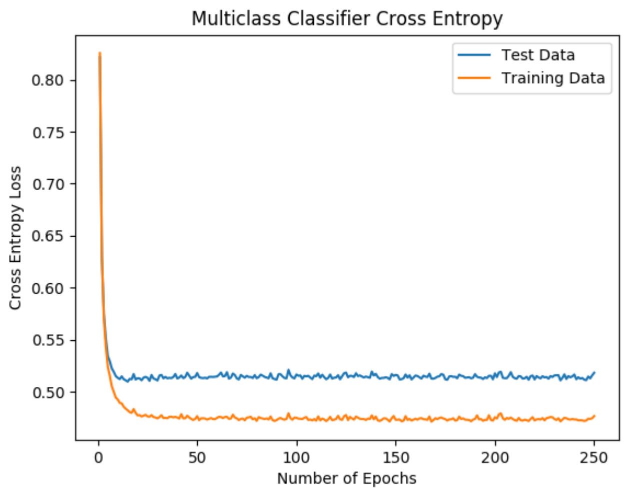


Figure 8: LMulticlass Classifier cross entropy



■

2 Neural Networks

2.1 Geometry of neural networks

Problem 2.1.1. *Consider learning a single sigmoid neuron to classify a linearly separable binary classification dataset (i.e. a hyperplane decision boundary can perfectly separate out the positive and the negative examples). Show that without weight decay, the length of optimal* *weights vector W learnt on this dataset will be unbounded. In other words, show that the L2 norm of the optimal weights $||W^*||_2 = \infty$.*

Solution. If the solution is linearly separable, $\exists$ a $\mathbf{x}$ that $W * \mathbf{x} + b = 0$ defines the hyper planes where all the points above the plane, i.e $W * \mathbf{x} + b > 0$, are positive(1), while all the points below the plane, i.e. $W * \mathbf{x} + b < 0$ are labelled as negative (0).
Analytically, the loss function incurred by the weight scheme discussed above would be as follows:

$$\mathcal{L} = \sum_{m=1}^{M} \frac{1}{M}\Bigg[- y^{(m)} log\hat{y}(x^{(m)}) - (1 - y^{(m)}) log(1 - \hat{y}(x^{(m)}))\Bigg]$$

Ultimately, some loss occurs if the labels or not correctly applied. As such, the optimal set of weights will correctly classify all points - a possible task giving the separability of the dataset. These weights effectively transform inputs into the single sigmoid function such that it acts as a signum function, with the separable hyperplane acting as the transition point of signum. That is, all $\mathbf{x}$ above the plane should be transformed by the weights to be equal to ∞, and all points below the plane should be transformed into $-\infty$. The sum of these weights thus must be infinite, and the sum of a finite series going to infinity requires that at least one of the members (weights) is also infinite. Thus, the L2 norm, the squared sum of these weights, must be equal to infinity. ■

Problem 2.1.2. *Consider learning a single sigmoid neuron to classify a linearly inseparable binary classification dataset (i.e. a hyperplane decision boundary can perfectly separate out the positive and the negative examples). Show that without weight decay, the length of optimal weights vector W learnt on this dataset will always be bounded. In other words, show that the L2 norm of the optimal weights $||W^*||_2 < \infty$.*

Solution. Consider the data set in Problem 2.1.1, with at st one training example that is inseparable (e.g. a positive point below the hyperplane, or a negative point above the hyperplane). As the dataset trains, it will also attempt to approach the same hyperplane separation that was discussed in the previous problem.

However, at a certain point, the the loss incurred by the incorrectly classifying the inseparable training example(s) will be greater than the loss removed by increasing the weights towards infinity. Moreover, the cross-entropy loss function tends towards infinity as the any point is completely incorrectly classified. As such, the utilization of infinite weights for a single sigmoid function in an inseparable case will ultimately generate infinite loss, and any finite set of weights is more optimal.

Thus the trained set of optimal weights, W^*, must be finite in both magnitude and length (given a finite set of dimensions), and as such, the L2 norm of the weights, $||W^*||_2^2 < \infty$, and is bounded. ■

Problem 2.1.3. *Consider learning a single hidden layer neural network to classify a linearly inseparable binary classification dataset. For all the weight matrices in the neural network, we can first flatten each weight matrix into a vector and let $vec\{w\}$ denote the vector that is the concatenation of all the flattened weight matrices. Under cross-entropy loss without weight decay, give constructive examples to show that some locally optimal solutions have bounded weight vectors, and that some other locally optimal solutions have unbounded weight vectors.*

Solution. It is assumed that this is a single hidden layer NN with a ReLU activation unit after the hidden network, output to a sigmoid output neuron. Consider the two-neuron hidden layer system - in order to simulate the behaviour of a single sigmoid neuron, to achieve the optimum of the situation in Problem 2.2, the weights can be scaled such that one neuron handles the positive portion of the curve while the other neuron handles the negative portion of the sigmoid curve, given that the ReLU function truncates the function at 0, meaning a single neuron under ReLU can only add positively or negatively to the final sigmoid output neuron. Given that the same situation is bounded in Problem 2.1.2, it is clear that the respective weights in this situation mus be bounded.

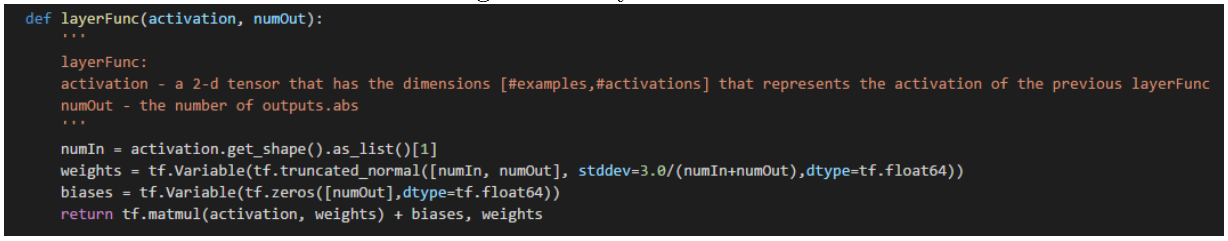
Given that there is a ReLU output layer, if a particular neuron must not be activated, it is possible that the weights must tend towards negative infinity, to prevent any sort of activation of the neuron. Consider a three neuron activation layer, where the first two neurons have exactly opposite output weights, and let us consider that the first two neurons have found the locally optimal solution afforded two neurons, as mentioned above. Adding a third neuron, it is in its best interest to never activate - it is possible for the weight of the input neurons will to go to negative infinity, in order to never activate the ReLU function. ■

2.2 Feedforward fully connected neural networks

Problem 2.2.1. ***layer-wise building block:*** *Write a vectorized Tensorflow Python function that takes the hidden activations from the previous layer then return the weighted sum of the inputs(i.e. the **z**) for the current hidden layer. You will also initialize the weight matrix and the biases in the same function. You should use Xavier initialization for the weight matrix. Your function should be able to compute the weighted sum for all the data points in your mini-batch at once using matrix matrix multiplication. It should not contain loops over the training examples in the mini-batch. The function should accept two arguments, the input tensor and the number of the hidden units. Include the snippets of the Python code.*

Solution. The function follows in the figure below:

Figure 9: Layer function



```
def layerFunc(activation, numOut):
    '''
    layerFunc:
    activation - a 2-d tensor that has the dimensions [#examples,#activations] that represents the activation of the previous layerFunc
    numOut - the number of outputs.abs
    '''
    numIn = activation.get_shape().as_list()[1]
    weights = tf.Variable(tf.truncated_normal([numIn, numOut], stddev=3.0/(numIn+numOut),dtype=tf.float64))
    biases = tf.Variable(tf.zeros([numOut],dtype=tf.float64))
    return tf.matmul(activation, weights) + biases, weights
```

Note that due to a misreading of the instructions, this layer function is not the same as in questions after section 2.2 - 4 arguments were originally used, and re done in 2.2 as a proof of concept. ■

Problem 2.2.2. ***Learning:*** *Use your function from the previous question to build your neural network model with ReLU activation functions in TensorFlow and tf.nn.relu can be useful. For training your network, you are supposed to find a reasonable value for your* *learning rate. You should train your neural network for different values of learning rate and choose the one that gives you the fastest convergence in terms of the training loss function. (You might want to babysit your experiments and terminate a particular run prematurely as soon as you find out that the learning rate value is not very good.) Trying 3 different values should be enough. You ay also find it useful to apply a small amount of weight decay to prevent overfitting. (e.g. $\lambda = 3e - 4$.) On the training set, validation set and test set, record your classification errors and cross-entropy losses after each epoch. Plot the training, validation, and test classification error vs. the number of epochs. Make a second plot for the cross-entropy loss vs. the number of epochs. Comment on your observations*

Solution. Using a weight decay λ of 3e-4, a batch size of 500, 100 training epochs and a one-hot encoding, a soft-max single layer neural network was implemented. In order to investigate the effects of learning rate, the training, validation, and test classification error was investigated as a function of training rate over several epochs. The cross entropy was also plotted as a function of learning rate over all epochs. Initial candidates for analysis were learning rates of η = [1e-1, 1e-2, 1e-3, 1-4, 1e-5] - by examining the training cross entropy loss and classification error(Figures 10-12), this was narrowed down into a list of [1e-2, 1e-3, 1e-4].

Figure 10: Training Cross Entropy vs Epoch

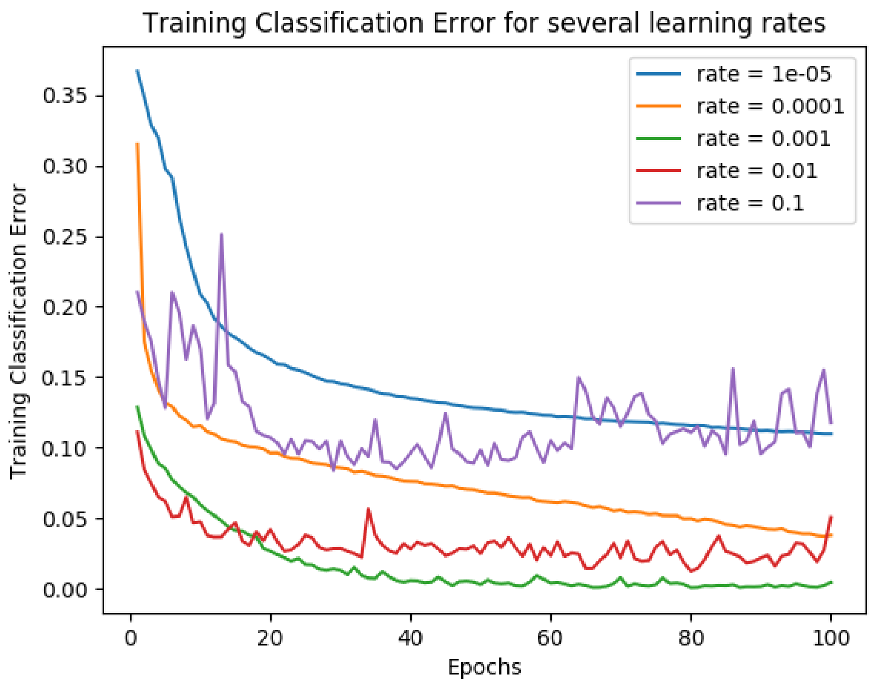


Figure 11: Training Classification error rate vs Epoch

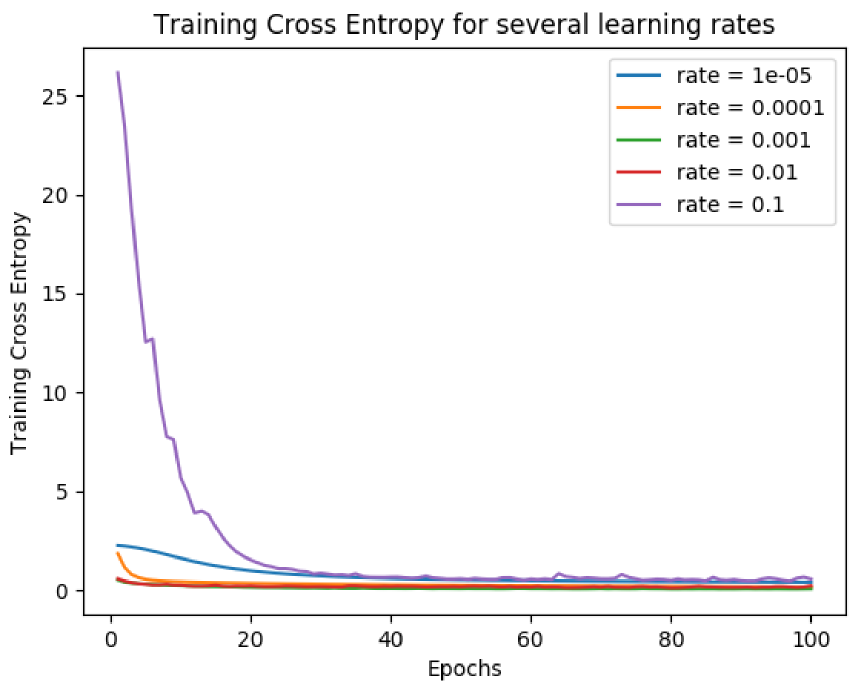
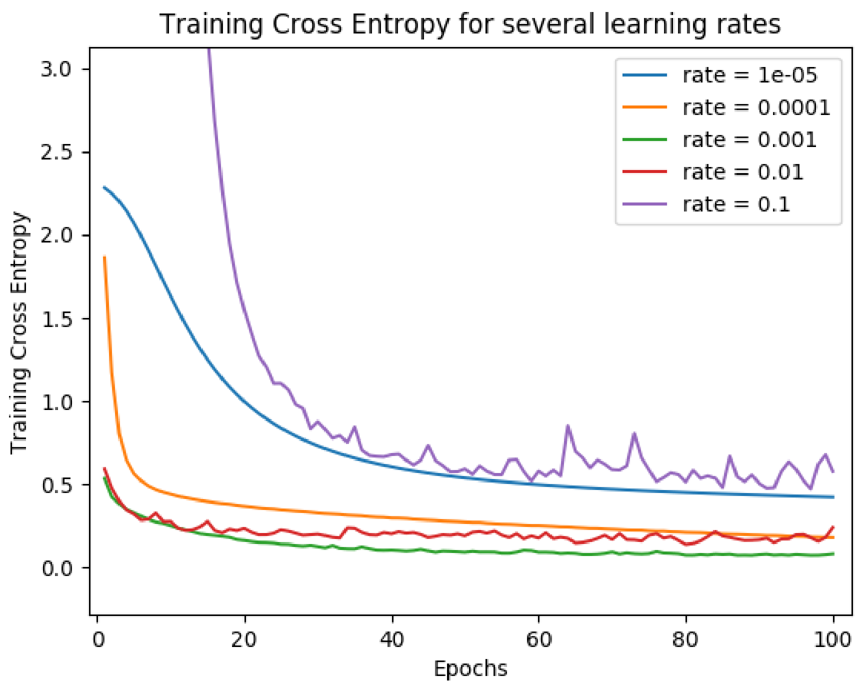


Figure 12: Training Classification error rate vs Epoch, zoomed



For the learning rates of choice, the classification error and cross entropy was examined across all epochs, as shown in Figures 13-18.

Figure 13: Cross entropy vs epoch, rate = 1e-4

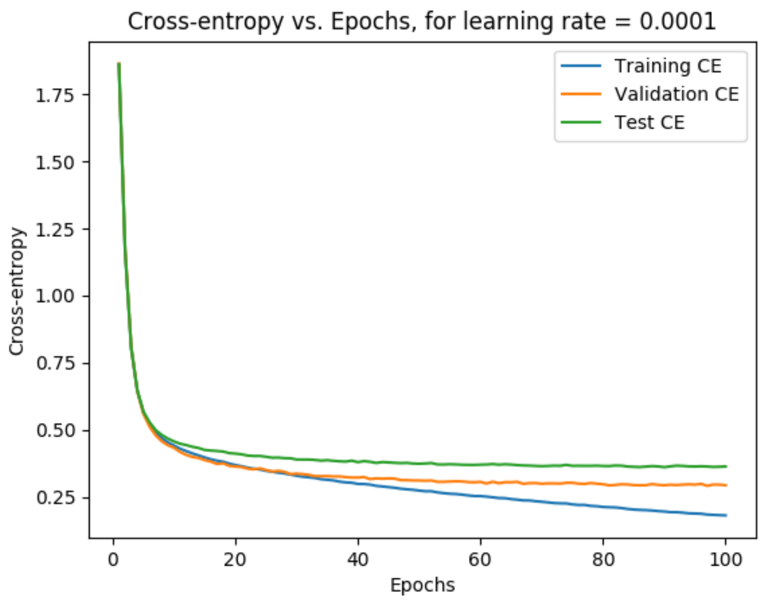


Figure 14: Classification error vs epoch, rate = 1e-4

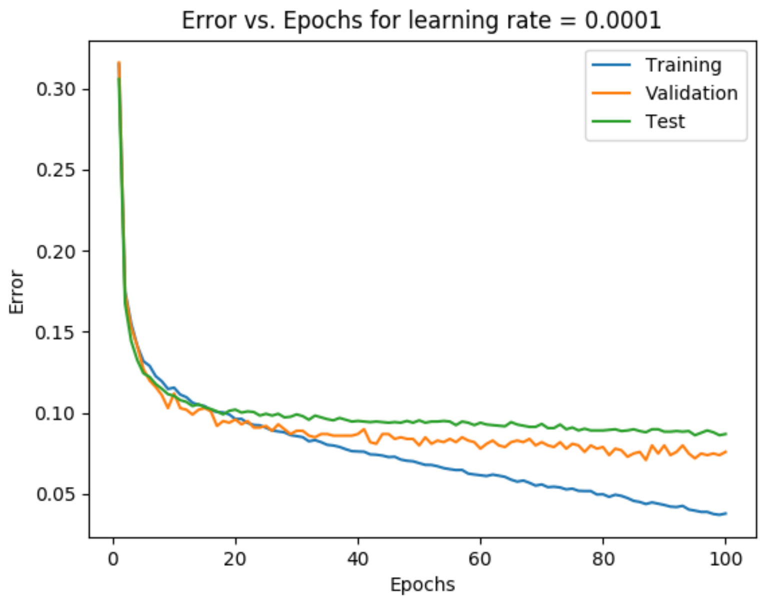


Figure 15: Cross entropy vs epoch, rate = 1e-3

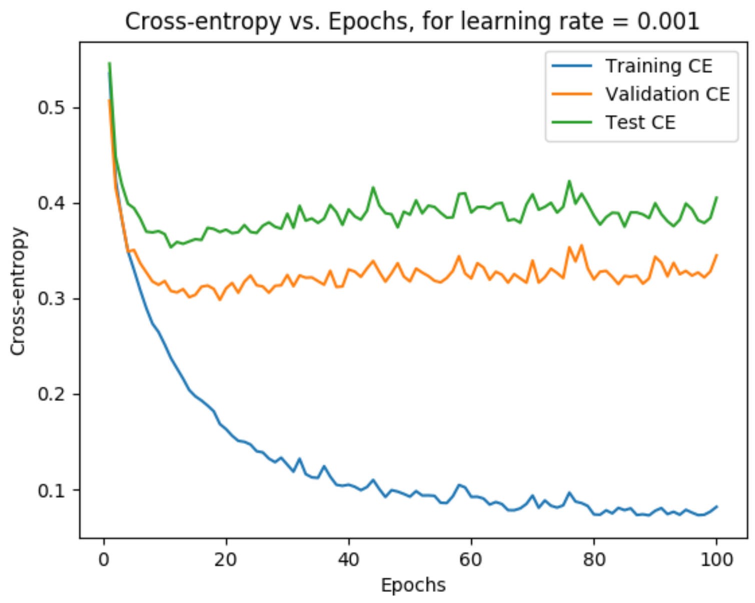


Figure 16: Classification error vs epoch, rate = 1e-3

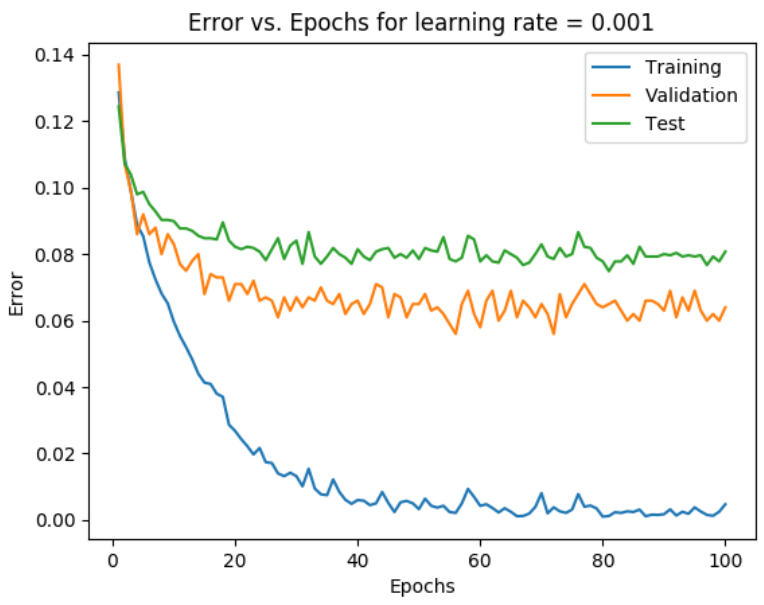


Figure 17: Cross entropy vs epoch, rate = 1e-2

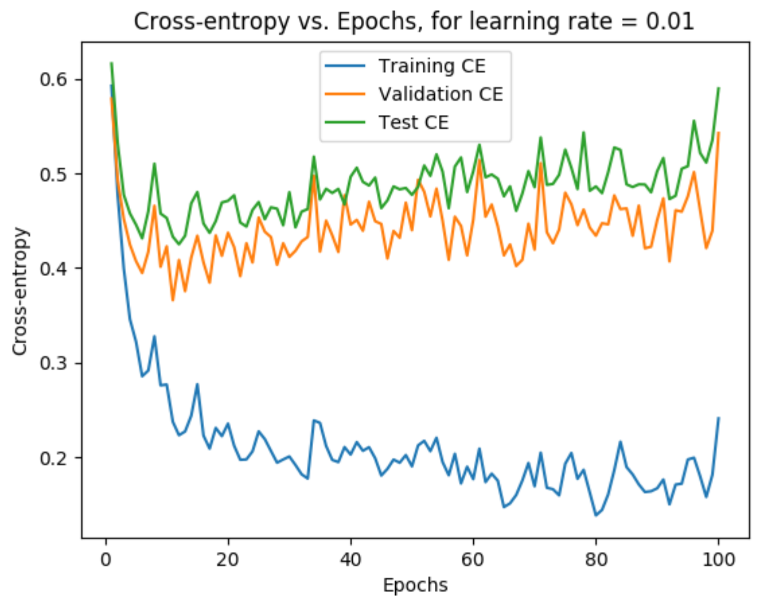
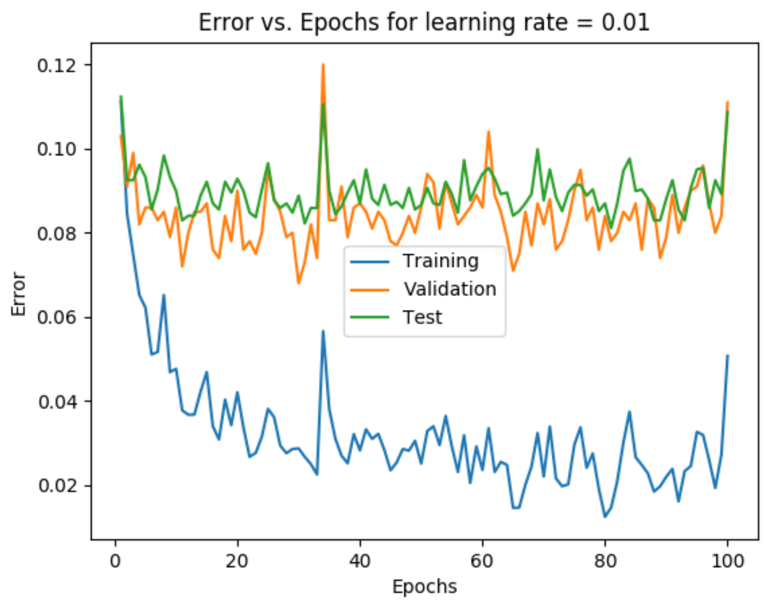


Figure 18: Classification error vs epoch, rate = 1e-2



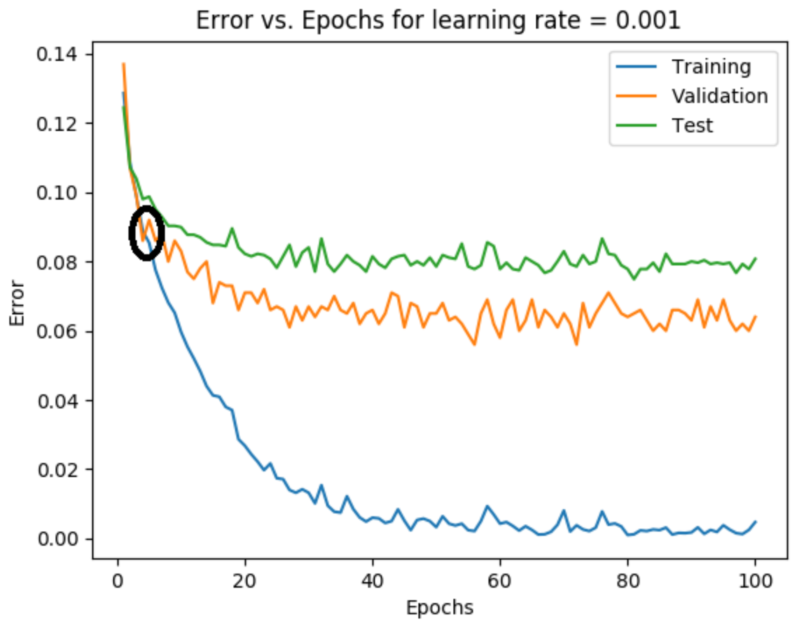
From these graphs, of the 5 learning rates, 1e-3 is the most reasonable learning rate, offering quick convergence for training cross-entropy, and the highest generalizability, i.e. lowest average errors for both validation and test data. Learning rates lower than this rate fail to converge in sufficient time(Figure 13), while learning rates greater than 1e-3 tend to have high oscillations(Figure 17,18) in accuracy, and tend to converge to higher losses.

■

Problem 2.2.3. ***Early stopping:*** *Early stopping is the simplest procedure to avoid overfitting. Determine and highlight the early stopping point on the classification error plot from question 2.2.2, and report the training, validation and test classification error at the early stopping point. Are the early stopping points the same on the two plots? Why or why not? Which plot should be used for early stopping, and why?*

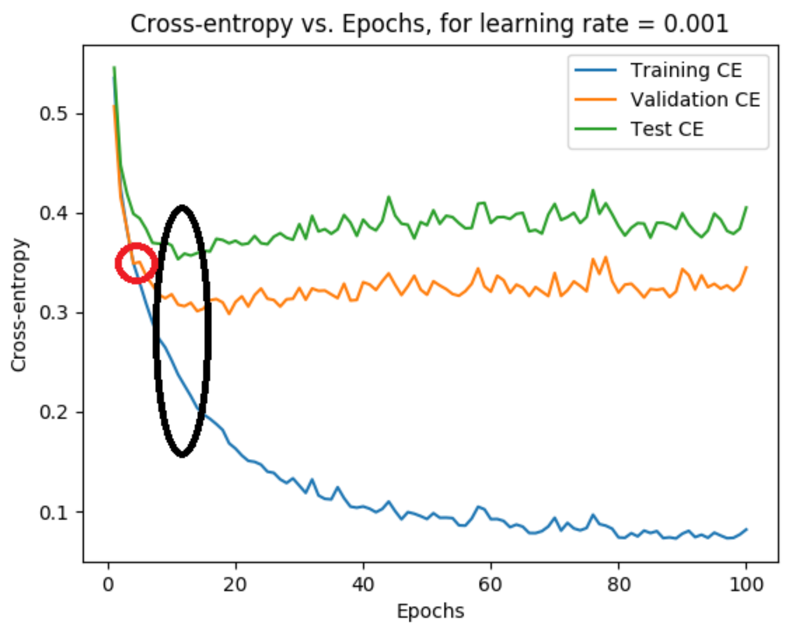
The early stopping point for the optimum cases (rate = 1e-3) using the classification error plot is after epoch 5, where a sharp jump in classification error would stop the algorithm. This is highlighted in Figure 19. At this point, the training error is 0.085, validation error is 0.093, and the test error is 0.10.

Figure 19: Early stopping due to classification error



As a alternative perspective, the early stopping point from the cross entropy is highlighted in red in Figure 20. Examining minute trends, the first eligible early stopping point occurs at the tth epoch during training. At this point, errors are as previously stated. In comparison, the early stopping point examining the larger trend (in black) occurs around epoch 15. In this case, At this point, the training error is 0.04, validation error is 0.07, and the test error is 0.08.

Figure 20: Early stopping due to cross entropy



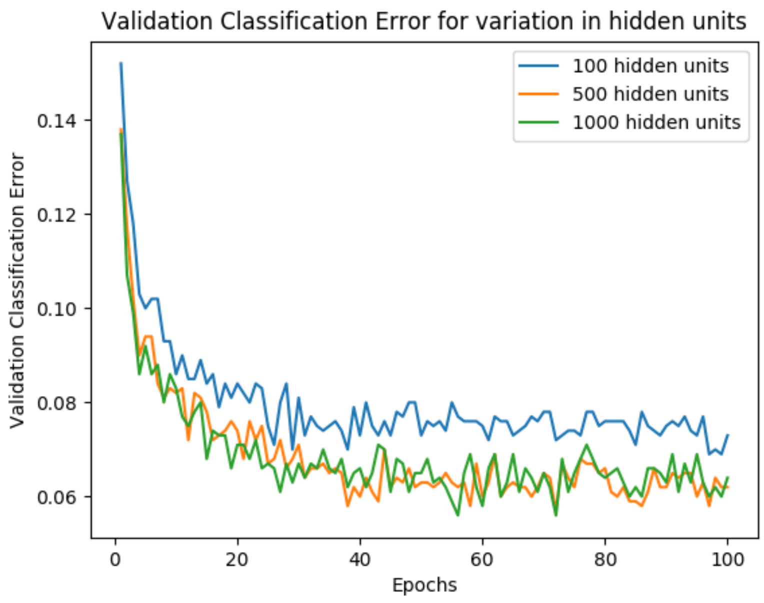
The first early stopping point is identical for the two plots, at epoch 4. Given that the classification error can be treated as a quantized version of the cross entropy, this is unsurprising. However, the cross entropy should still be used for early stopping - the fact that the statistic remains unquantized allows for a more sensitive extermination of possible overfitting. Indeed - there is a larger "global" minima that is obvious in the cross entropy plot that is completely invisible in the classification error plot.

2.3 Effect of hyperparameters

Problem 2.3.1. ***Number of hidden units***: *Instead of using 1000 hidden units, train different neural networks with [100, 500, 1000] hidden units. Find the best validation error for each one. Choose the model which gives you the best result, and then use it for classifying the test set. What is the number of test errors (or test classification errors)? In one sentence, summarize your observation about the effect of the number of hidden units on the final results.*

Solution. The validation errors for each number of hidden units was examined across 100 epochs. The results are shown in Figure 21.

Figure 21: Early stopping due to cross entropy

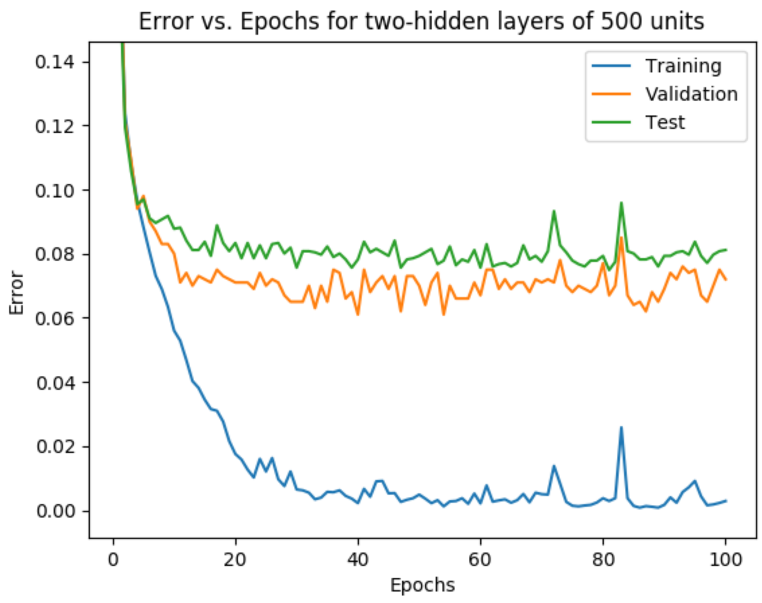


The two best models were those 500 and 1000 hidden units (nigh indistinguishable from the graph above). The number of test classification errors at the end of 100 training epochs was 205 and 220 errors out of 2724 test points respectively. Ultimately, judging by the validation error rate and test errors, it seems that having too many hidden units causes overfitting, while too little hidden units results in insufficient flexibility to properly model the system. ■

Problem 2.3.2. ***Number of layers****: For this task, train a neural network with two hidden layers of 500 hidden units each (1000 total). Plot the number of training and validation errors (or training and validation classification errors) vs. the number of epochs. What is the final validation error when training is complete? Using the test set, compare this architecture with the one-layer case.*

Solution. The validation and classification error rate for the new architecture was examined across 100 epochs. The results are shown in Figure 22.

Figure 22: Classification error vs epochs for a two-layer architecture



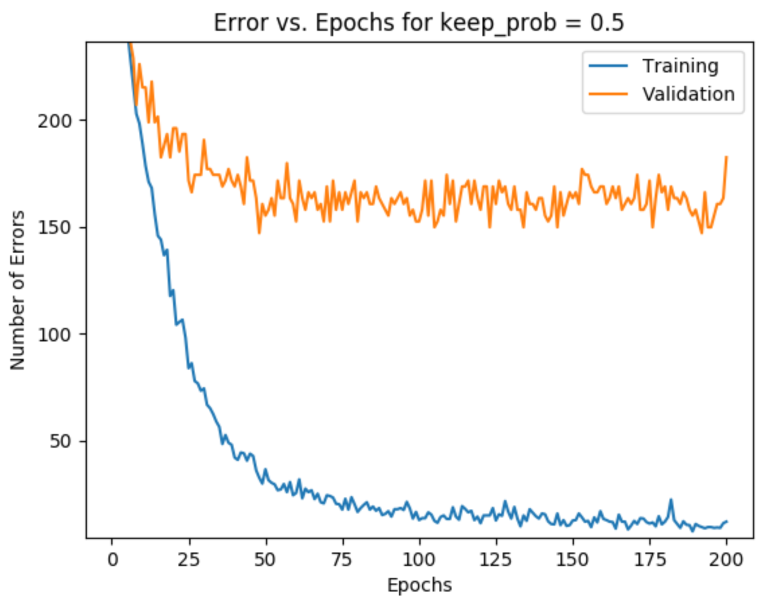
We can compare the effect of the number of layers with Figure 8. All in all, there is a general increase in noise in the classification error, and the mean classification error has increased a slight bit (from rates of 0.065 to 0.07 on average for validation, and from rates of 0.08 to 0.085). This makes sense - larger number of layers allow for further abstraction, and consequently, overfitting, despite an applied weight decay. ■

2.4 Regularization and visualization

Problem 2.4.1. ***Dropout*** *: Introduce dropout on the hidden layer of the neural network (with dropout rate 0.5) and train you neural network. As you know, dropout should only be used in the training procedure, and the mean network should be used instead during the evaluation. Plot the number of training and validation errors (or training and validation classification errors) vs the number of epochs. Compare the results with the case that we do not have any dropout. Summarize the observed effect of dropout. (You may want to use the tf.nn.dropout utility function.)*

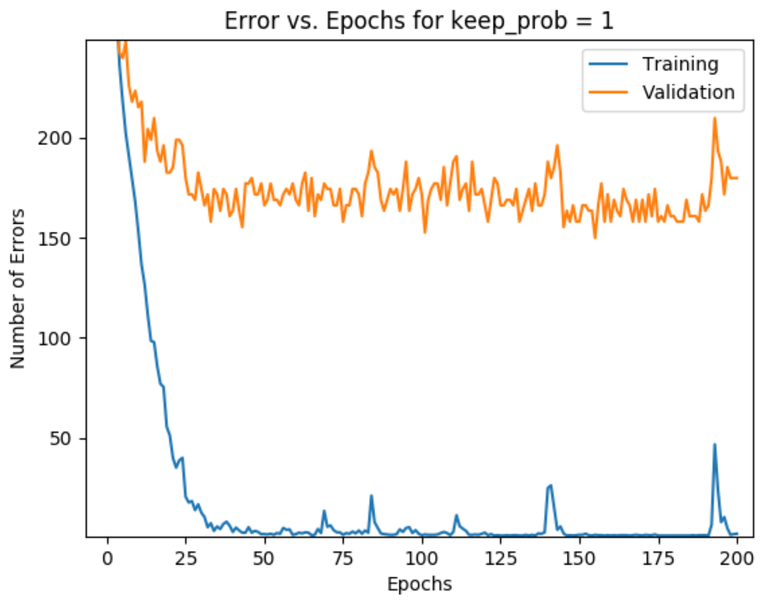
Solution. Drop out was introduced to the original network (Section 2.2), after the hidden layer before the output layer. The number of training and validation errors is plotted per epoch in Figure 23.

Figure 23: Classification error vs epochs for a network with dropout



This can be compared to a network with no dropout, i.e. a keep probability of 1 for each neuron. The same analysis is shown in Figure 24.

Figure 24: Classification error vs epochs for a network without dropout



On average, the training errors don't decay as fast with dropout implemented. However, the steady state error for validation has a noticible difference - the errors without dropout are on average around 170, while the errors with dropout are on average around 160. While the effect is minor, it is consistent, indicating that the effect of dropout is to properly regularize the networks such that it does not overfit the training data during learning.

In addition to these general statements, it should be noted that the early stopping point using cross-entropy loss can be shown to be around 12 epochs worth of training. ■

Problem 2.4.2. ***Visualization*** *: It is generally hard to figure out what the hidden neurons are doing in a neural network. However, one can visualize the incoming weight vector to each hidden neuron in the first hidden layer as an image and inspect what that neuron has learnt from the dataset. For the neural networks we have trained in this assignment, we can visualize the values of the 28x28 incoming weights of each hidden units in the first layer as a grey-scale image. For the models trained with and without dropout, checkpoint the model during training at 25%, 50%, 75% and 100% of the early stopping point. For each of the checkpoints, make a figure visualizing the 100 neurons in the first hidden layer. Comment on the learning progression of the hidden neurons, and comment on any difference between the dropout model and the one without dropout.*

Solution. As mentioned, the early stopping point occurs at 12 epochs. As such, snapshots of the learning with and without dropout are performed at various percentages of this metric. The coefficients for dropout and non-dropout cases are shown below, in Figures 25-32. Note that the first 100 weights are samples for each set of vectors, showing a incomplete sampling of information (for sanity and for brevity).

Figure 25: Visualization of the first 100 neurons at 25% training, with dropout

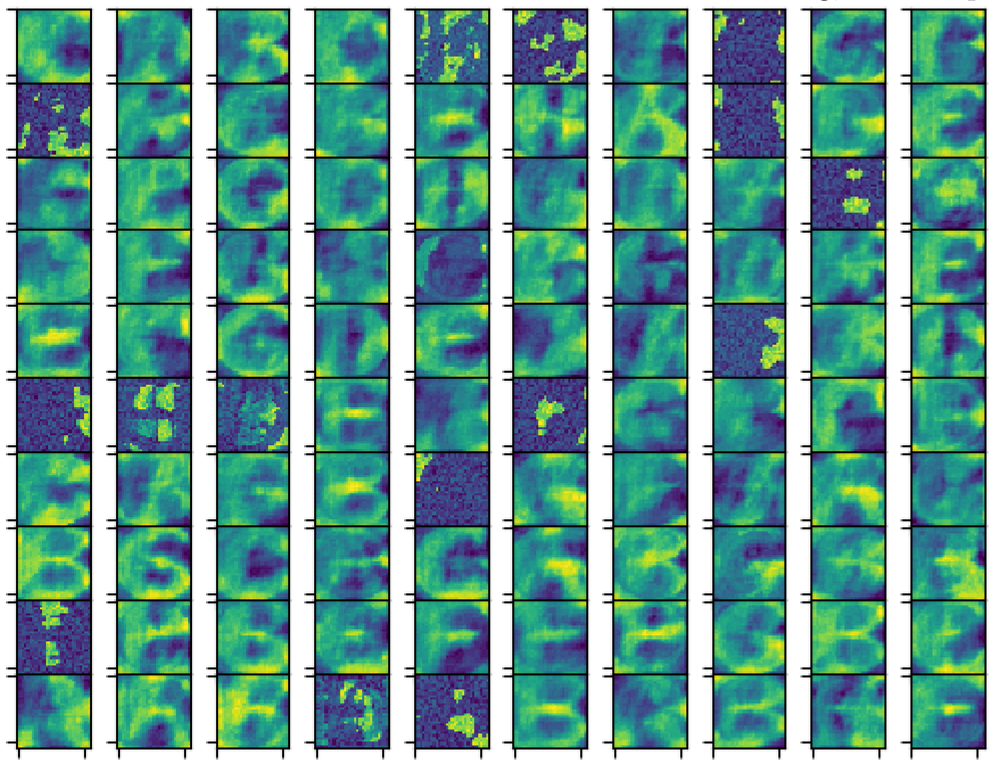


Figure 26: Visualization of the first 100 neurons at 50% training, with dropout

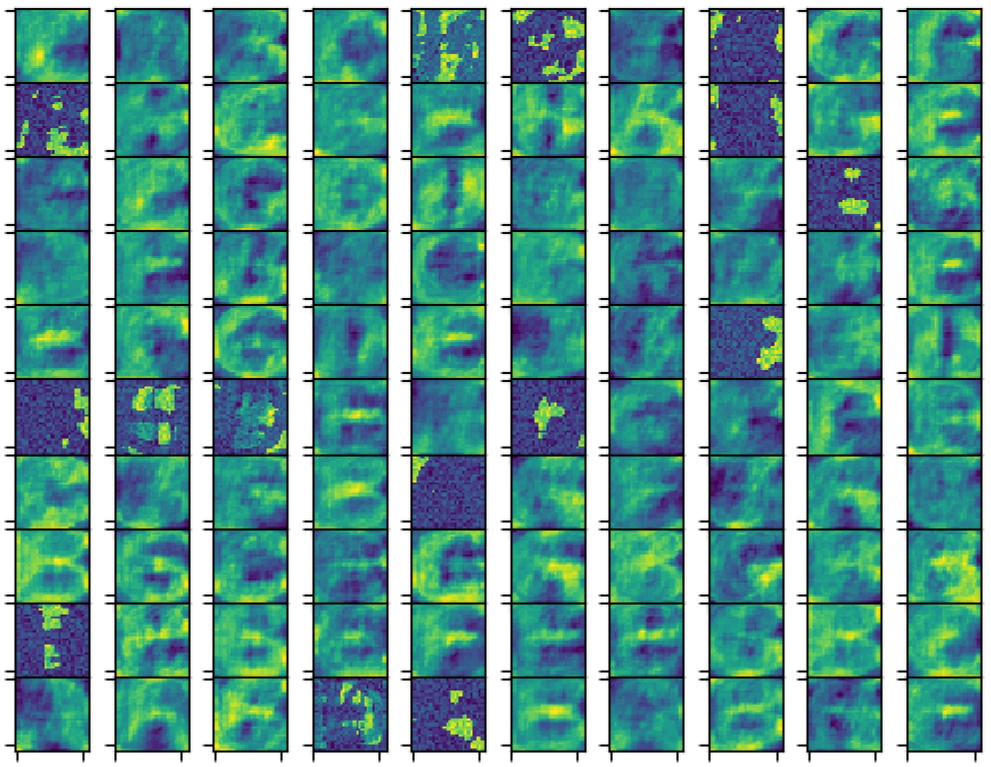


Figure 27: Visualization of the first 100 neurons at 75% training, with dropout

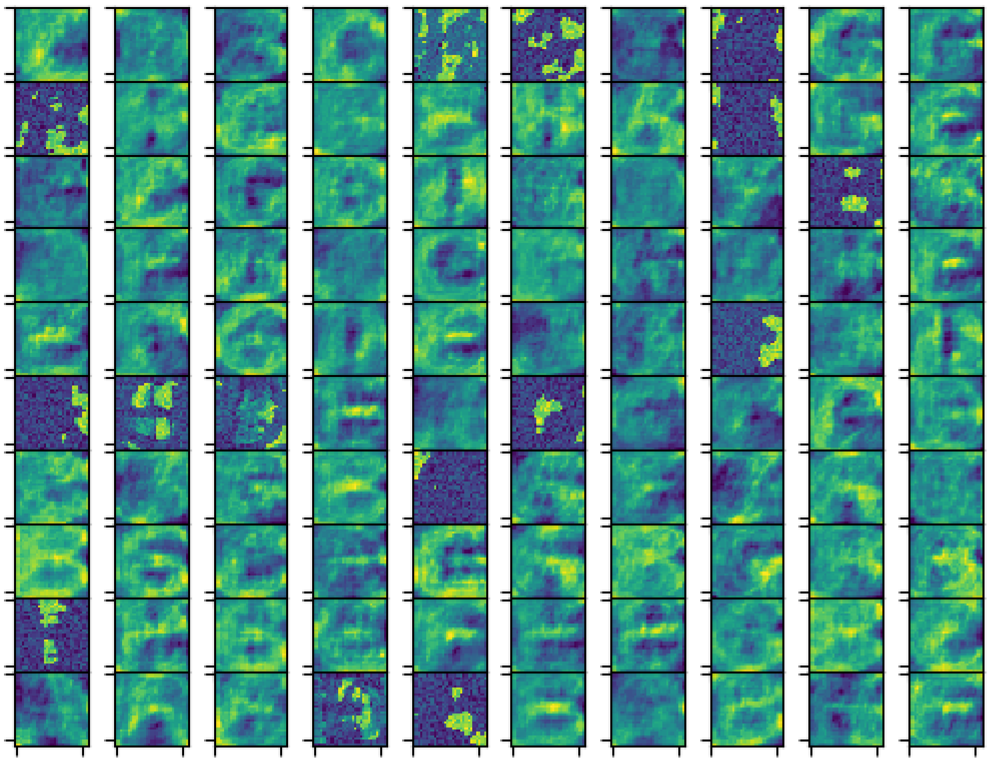


Figure 28: Visualization of the first 100 neurons at 100% training, with dropout

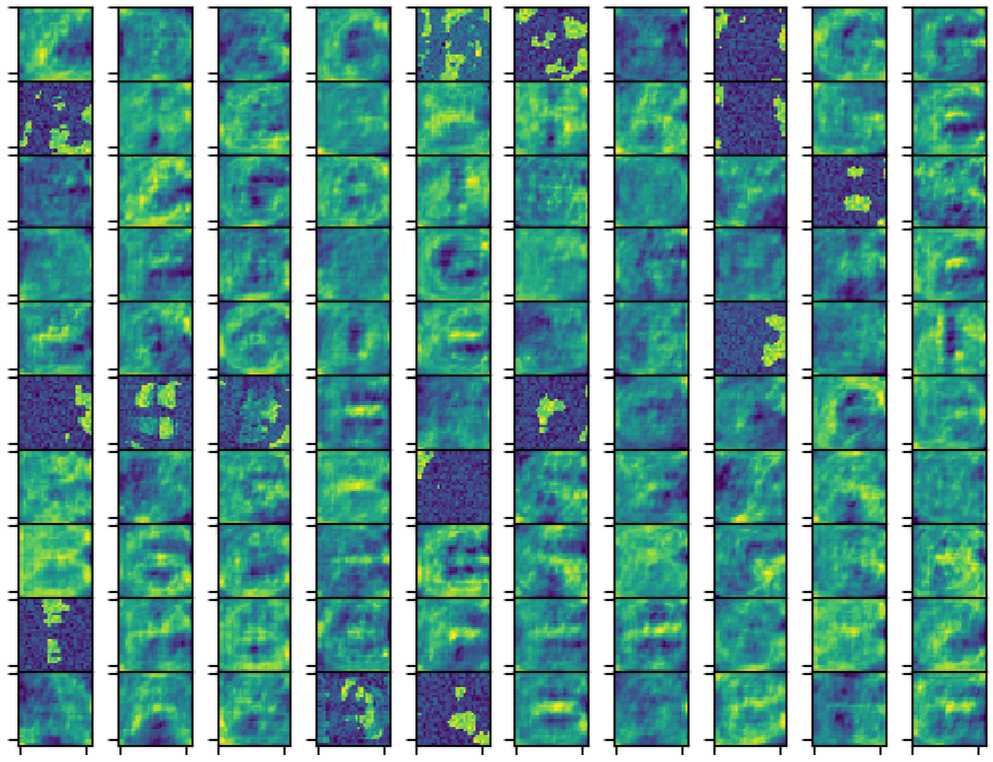


Figure 29: Visualization of the first 100 neurons at 25% training, without dropout

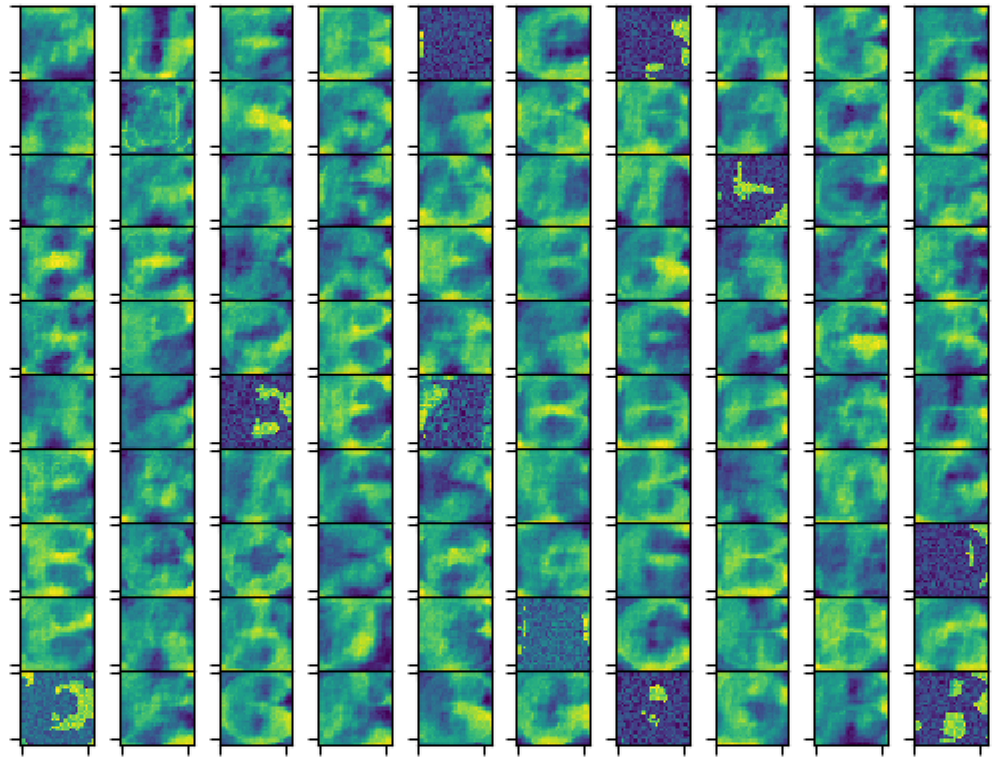


Figure 30: Visualization of the first 100 neurons at 50% training, without dropout

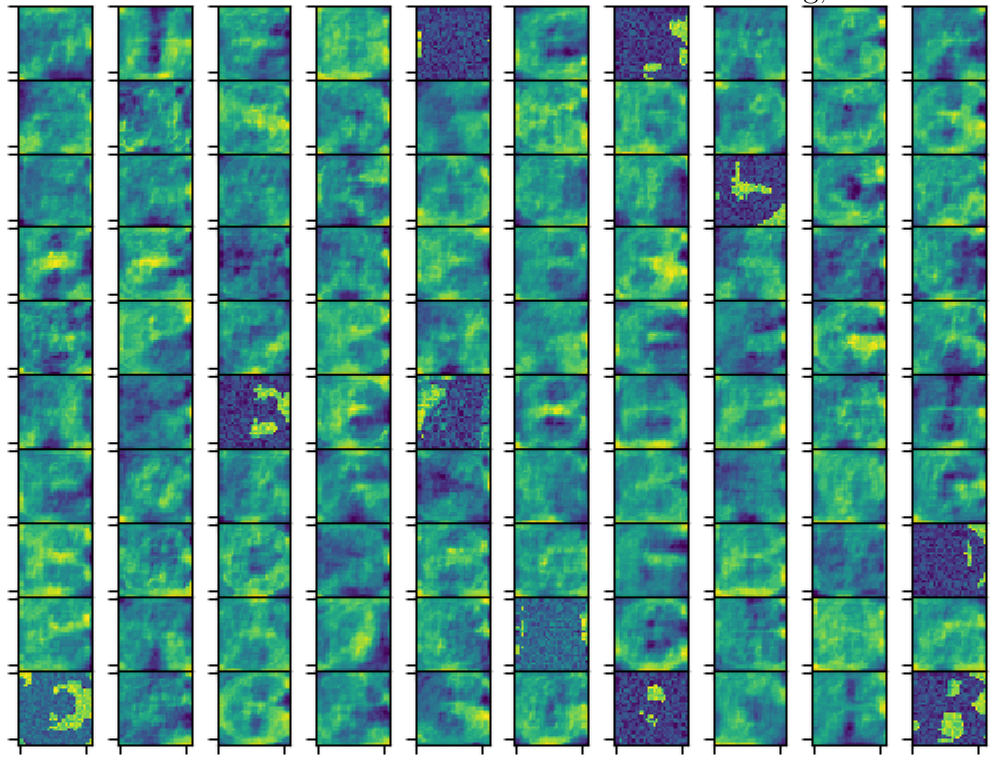


Figure 31: Visualization of the first 100 neurons at 75% training, without dropout

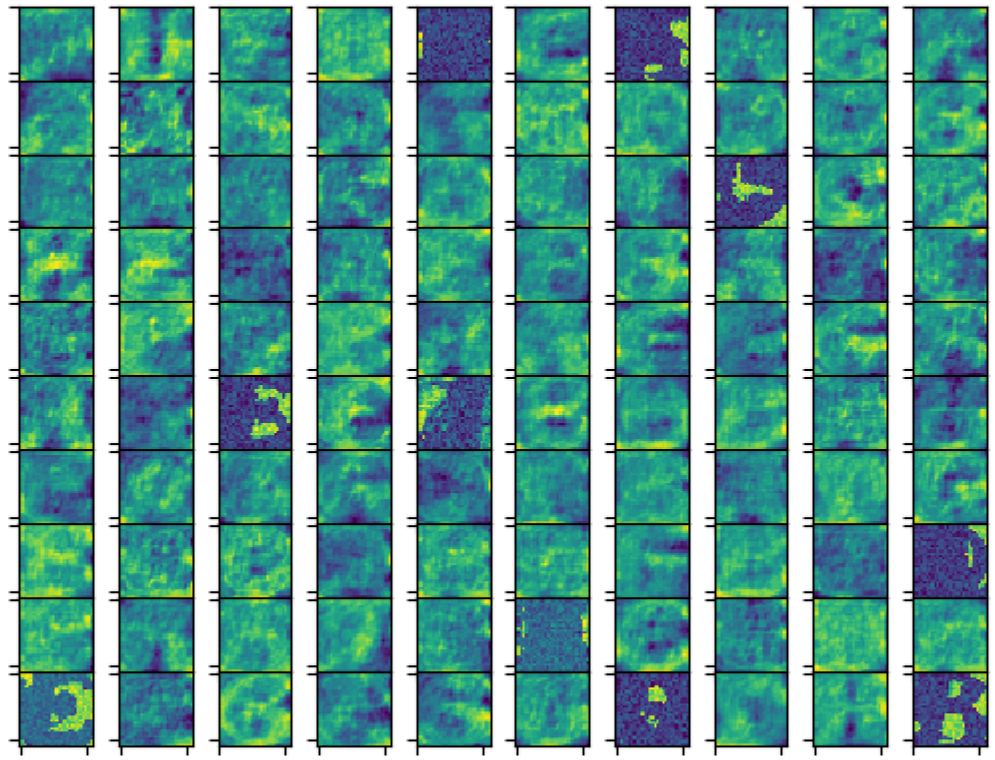
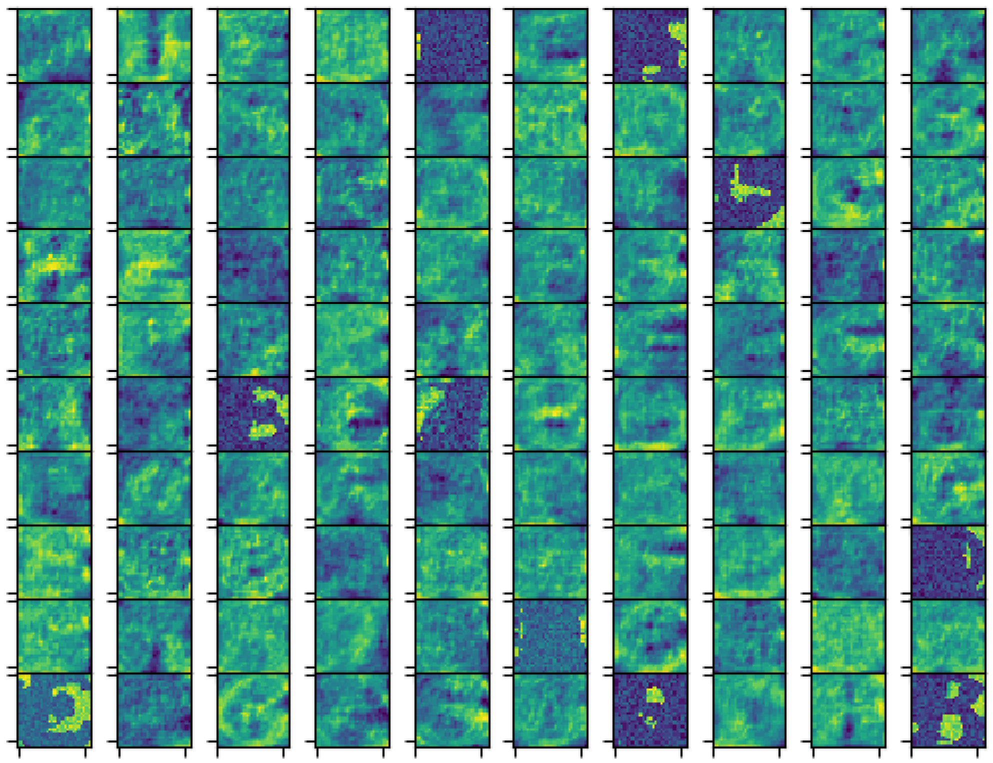


Figure 32: Visualization of the first 100 neurons at 100% training, without dropout



Initially, the hidden neurons are distinctly created from letters (from Figures 25 and 29, images of letters from A - J are clearly seen). However, as further training occurs, blurring occurs between letters, symbolizing some weighted contribution from all 10 possible classes. This is seen both in Figure 28, with dropout, and in Figure 32, without dropout.

The difference between the dropout and no-dropout models is best seen in Figures 28 and 32 - while the dropout model retains some strong signatures from various letters, there are a noticeable amount of neurons in the no-dropout model that simply either disappear (0 contribution from almost the entire image), or become a blurred summation of all pixels, revealing no novel information (3rd row, 1st to 3rd column of Figure 30). There are substantially less of these types of neurons in the dropout model, showing its performance as a regularizing operation. ■

2.5 Exhaustive search for the best set of hyperparameters

Problem 2.5.1. ***Random search*** :*First, try to set the random seeds in Numpy and TensorFlow libraries differently from other groups. (e.g. seed using the time of your experiments or some combination of your student IDs). Randomly sample the natural log of learning rate uniformly between -7.5 and -4.5, the number of layers from 1 to 5, the number of hidden units per layer between 100 and 500, and the natural log of weight decay coefficient uniformly from the interval [9, 6]. Also, randomly choose your model to use dropout or not. Using these hyperparameters, train your neural network and report its validation and test classification error. Repeat this task for five models, and report their results.*

Solution. The 5 set of tests had very specific random generation to increase reproducibility: All datasets were trained for 100 epochs, with mini batch sizes of 500. Random seeds for numpy were generated between 0 and 4294967295 using Python's built-in random module to randomize the neural network topology. For the purposes of data creation and training, the numpy seed was set to 1125, while the TensorFlow seed was set to 1127.

The 5 sets of experiments are given as follows:

Model	A	B	C	D	E
# of Hidden Layers	3	1	2	5	2
# of Hidden Units	[409, 258, 387]	[497]	[126, 296]	[260, 270, 454, 404, 366]	[413, 168]
Dropout	Yes, 0.5	Yes, 0.5	Yes, 0.5	No	Yes, 0.5
Weight Decay	1.155e-3	2.225e-3	1.382e-4	7.482e-3	1.911e-4
Learning Rate	1.093e-2	7.800e-4	3.516e-3	2.261e-3	2.395e-3
Validation Accuracy	0.885	0.925	0.918	0.908	0.928
Test Accuracy	0.885	0.932	0.919	0.923	0.929

■

Problem 2.5.2. ***Exchange ideas among the groups*** :*Collaboration is the key ingredient of major scientific progress and innovation. Advances in the fields of artificial intelligence and machine learning tend to depend on the rapid sharing of ideas and datasets. You will now exchange and compare your experimental results and the hyperparameter settings with other groups. (Yes, this means you will have to talk to other students in the class in person or use Piazza). You will now use the best hyperparameter you heard from someone who heard from someone else and report the best test error rate that can be achieved under the hyperparameter space in this question. Let us hope the wisdom the crowd will converge to a single best model.*

Solution. Thanks to the grapevine of Piazza, we now have various hyperparameters to choose from. The following parameters were chosen from a fellow student on Piazza:

- 2 hidden layers (247, 355)
- no dropout
- weight decay coefficient = 0.000805
- learning rate = 0.000516

The reported best results were of 0.94 and 0.925 validation and test results. When simulated under the 1125 and 1127 seeds for numpy and TensorFlow respectively, we report values of 0.924 and 0.922 for the validation and test classification results. These results are similar enough to show indicate that this classification algorithm is relatively robust. ■