Name: Supriyo Ghosh
Student ID: 215318728
Assignment: 2

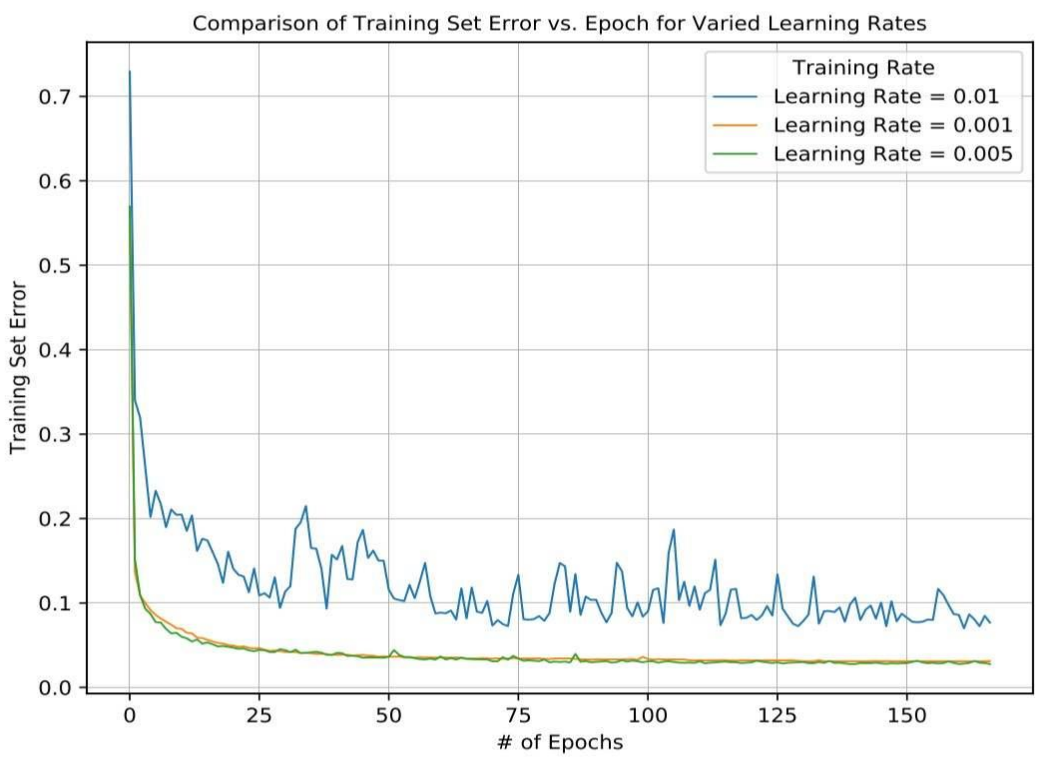
1.1 – Feedforward fully connected neural networks

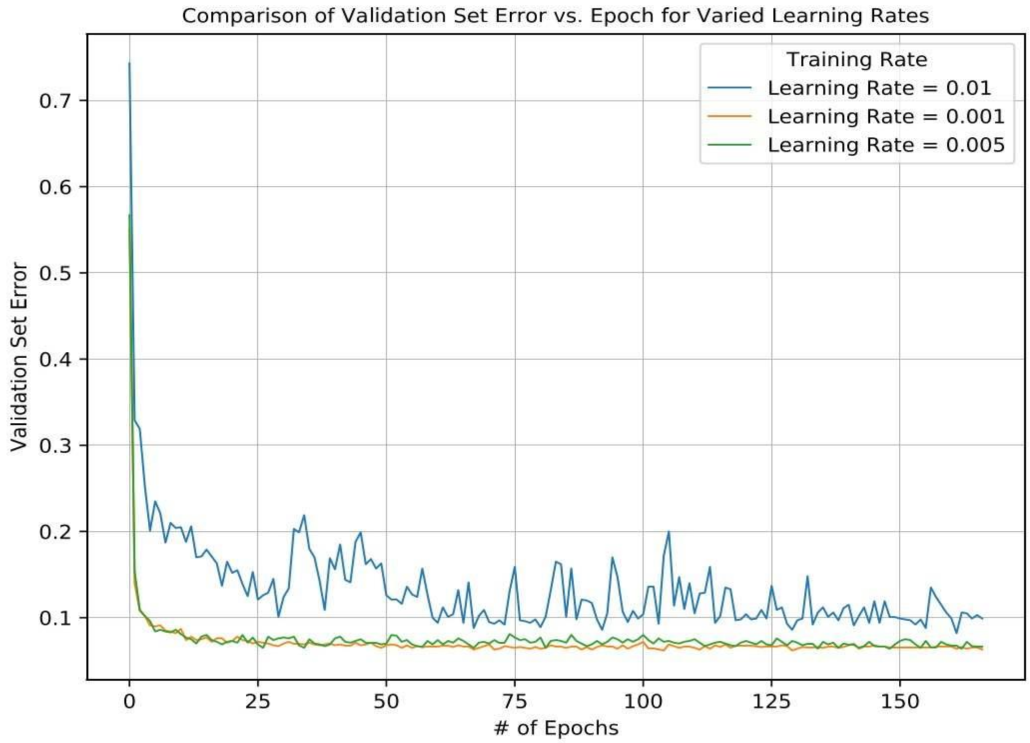
1.1.1 – Layer-wise building block

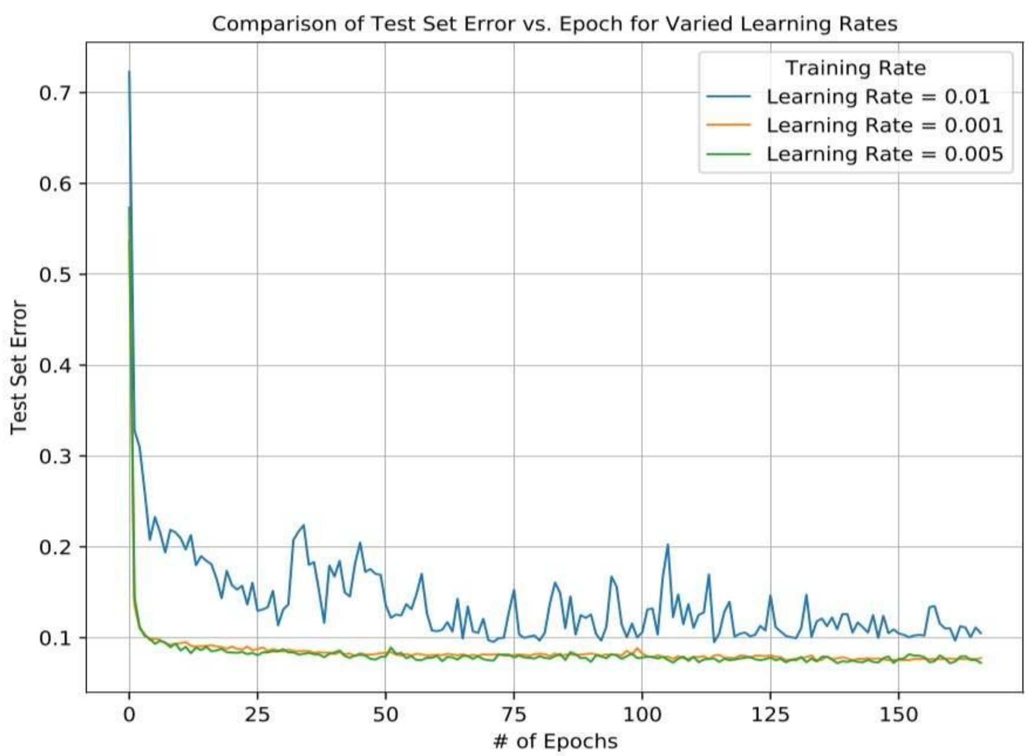
The *create_new_layer()* function creates a new neural network layer from the input tensor *input_tensor* with a specified number of hidden units. The input tensor represents the raw X input vector without the bias padding. The weight matrix is initialized using Xavier initialization, and the bias term is zero initialized.

1.1.2 – Learning

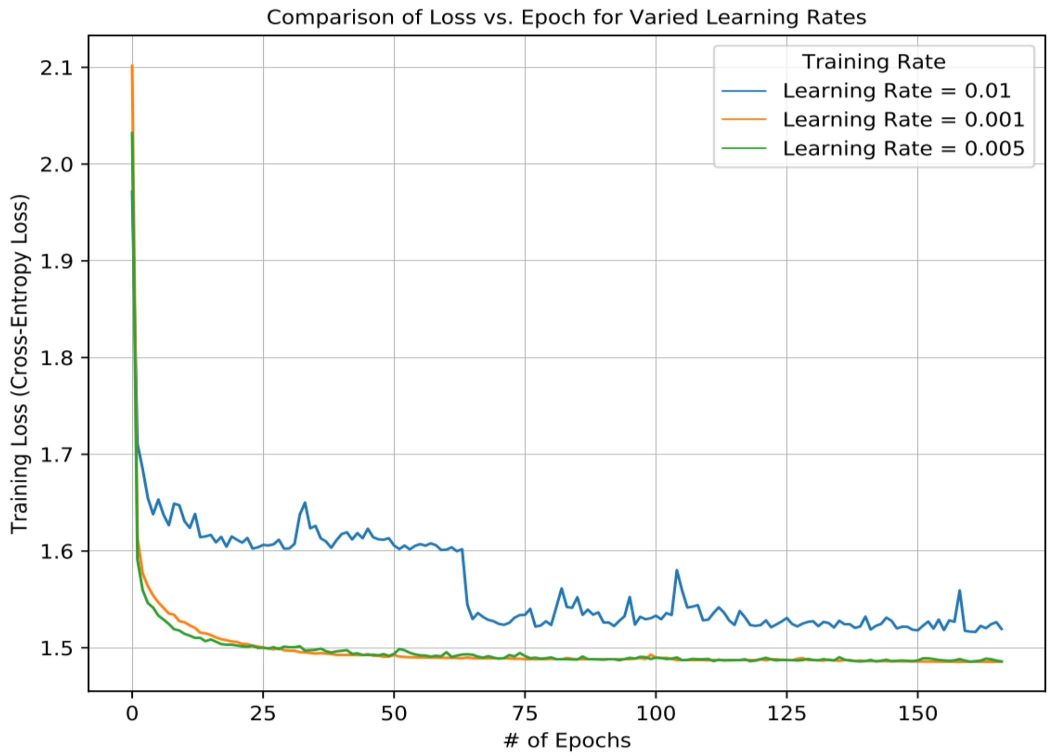
Using the above function, a simple one-layer neural network was created. To perform training, we used a minibatch size of 500 and 5000 iterations, totalling at 166 epochs. By setting the weight-decay coefficient $\lambda = 3 * 10^{-4}$, the following training, validation and test accuracy vs. epoch curves for different values of the learning rate η were observed:




Comparison of Validation Set Error vs. Epoch for Varied Learning Rates
Training Rate
Learning Rate = 0.01
Learning Rate = 0.001
Learning Rate = 0.005
Validation Set Error
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
25
50
75
100
125
150
# of Epochs


Comparison of Test Set Error vs. Epoch for Varied Learning Rates
Training Rate
Learning Rate = 0.01
Learning Rate = 0.001
Learning Rate = 0.005
Test Set Error
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
25
50
75
100
125
150
# of Epochs

Plotting the training set loss vs. epoch, the following curve was observed:



1.1.3 – Early stopping

Using the loss graph above and a learning rate of $\eta = 0.005$, we identify epoch 45 as the optimal early stopping point where loss did not appreciably change in later epochs. The test, training and validation accuracies at this point are recorded in the table below.

Training Set Error	Validation Set Classification Error	Test Set Classification Error
0.0352	0.075	0.0811

1.2 – Effect of hyperparameters

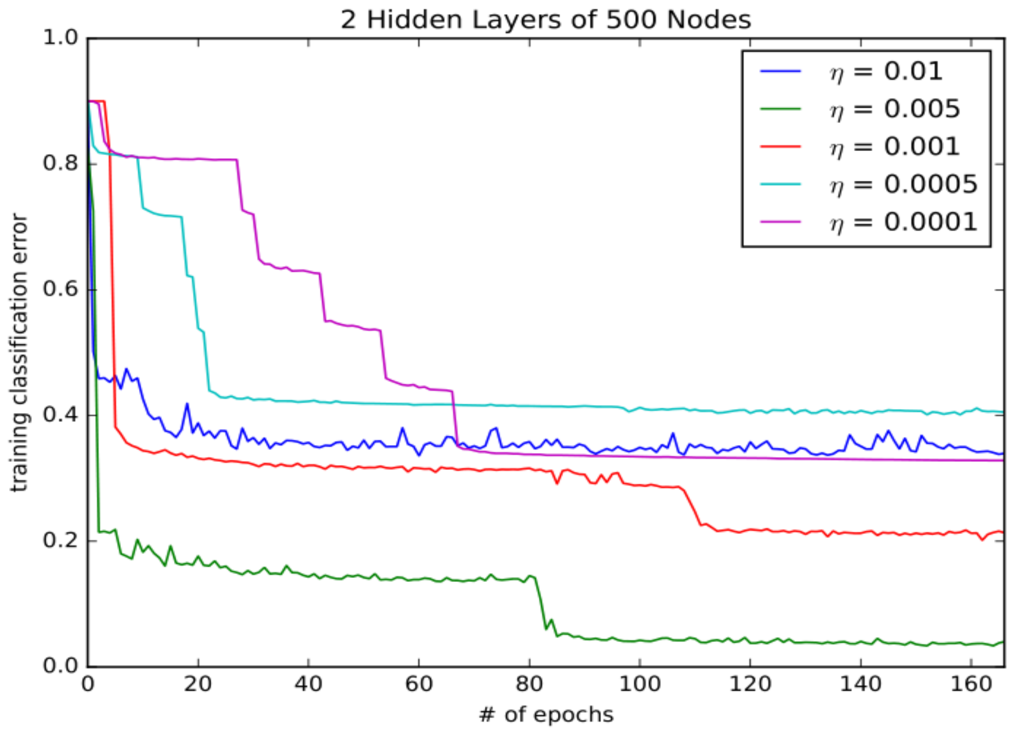
1.2.1 – Number of hidden units

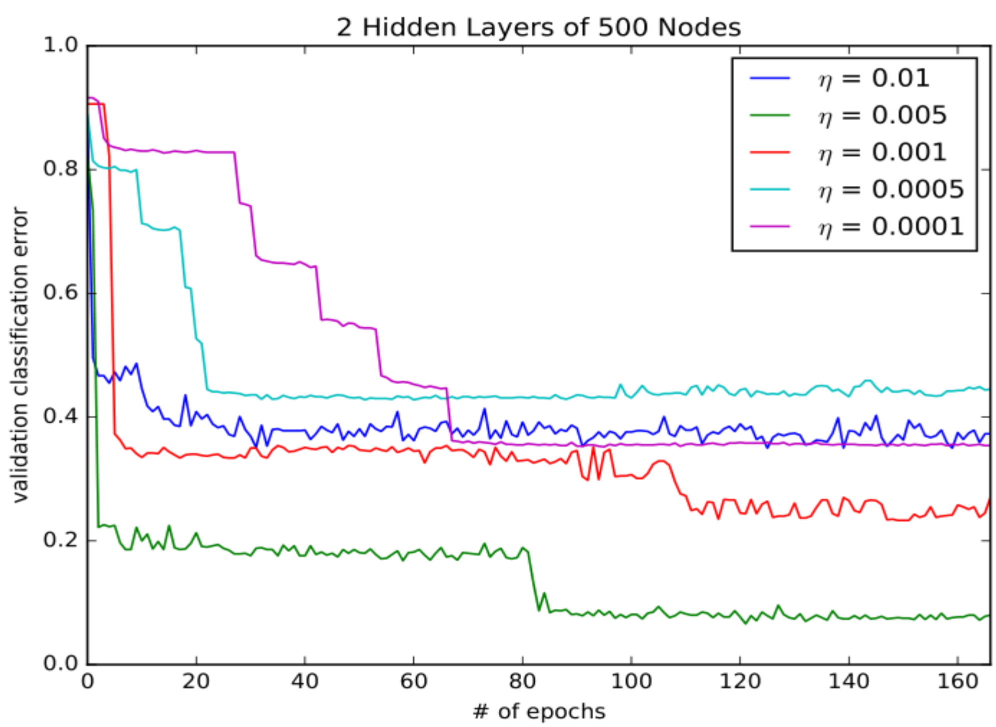
No weight decay was used in this model. The plots showing the training error and accuracies vs iterations (per epoch) can be seen in Appendix A. The following table summarizes the final values (best results bolded) for tuning the hyperparameters (where $\text{classification error} = 1 - \text{accuracy}$):

# of nodes	100			500			1000		
$\boldsymbol{\eta}$	0.01	0.005	**0.001**	0.01	0.005	**0.001**	0.01	0.005	**0.001**
Val. error	0.0800	0.0790	**0.0710**	0.0701	0.0670	**0.0670**	0.0760	0.0710	**0.0560**
Test error	0.0932	0.0870	**0.0896**	0.0808	0.0775	**0.0756**	0.0914	0.0793	**0.0727**

In summary, increasing the number of hidden units slightly reduces the validation error (by ~1%) for a single layer neural network for this dataset, but a lower number of nodes reduces computation time significantly.

1.2.2 – Number of layers





We can see in the graphs steep steps in decrease of error taking place at different epochs indicative of a multi-layer neural network. The following table summarizes the validation and test classification error final values for all learning rates. No weight decay was used in this model.

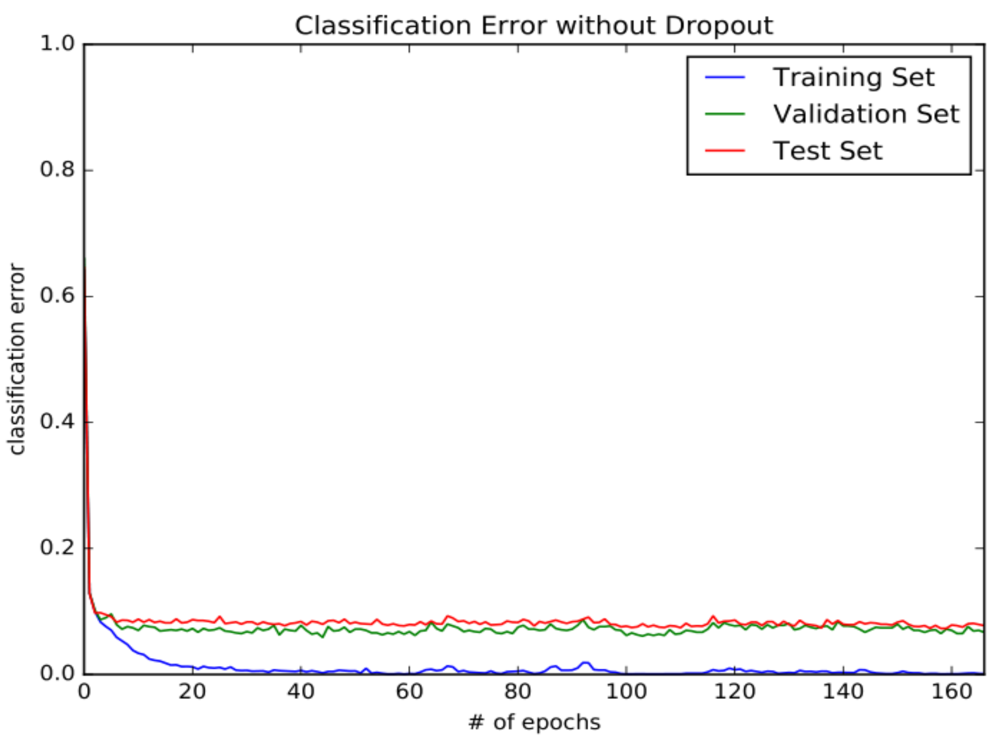
η	0.01	**0.005**	0.001	0.0005	0.0001
Val. error	0.373	**0.0800**	0.270	0.445	0.354
Test error	0.358	**0.0830**	0.262	0.452	0.356

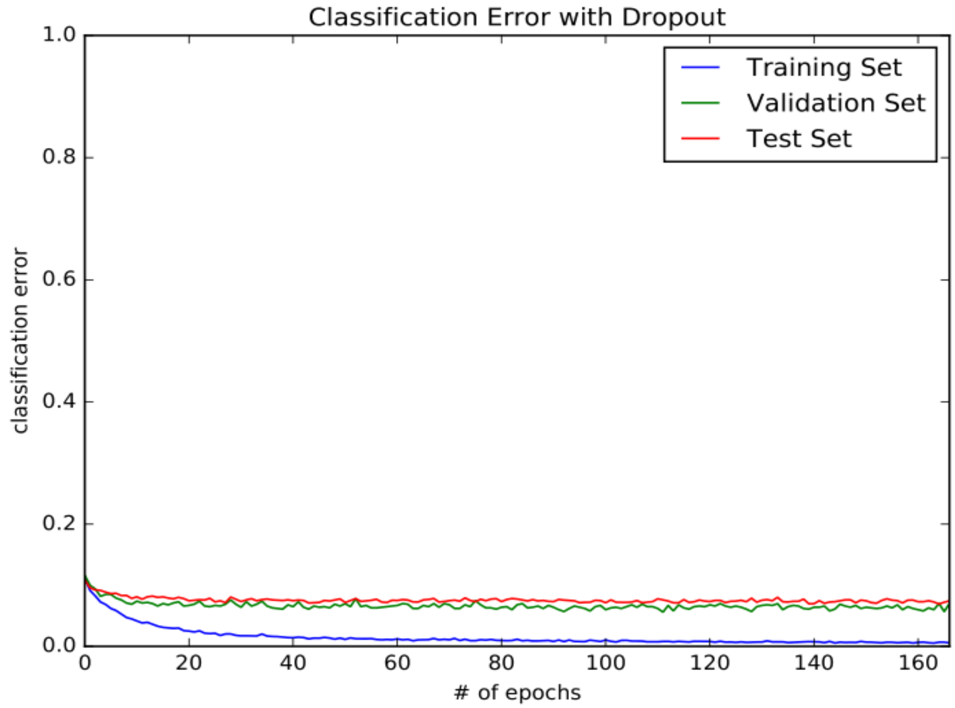
Comparing the results of the test set for $\eta = 0.005$ in the 2-layer neural network, we see that the error is on par, if not a bit worse than the single layer neural network.

1.3 – Regularization and visualization

1.3.1 – Dropout

The following are the resulting plots. As seen, adding dropout to the single layer neural network slightly increases test accuracy (i.e. lowers test classification error) but has a slower rate of convergence.

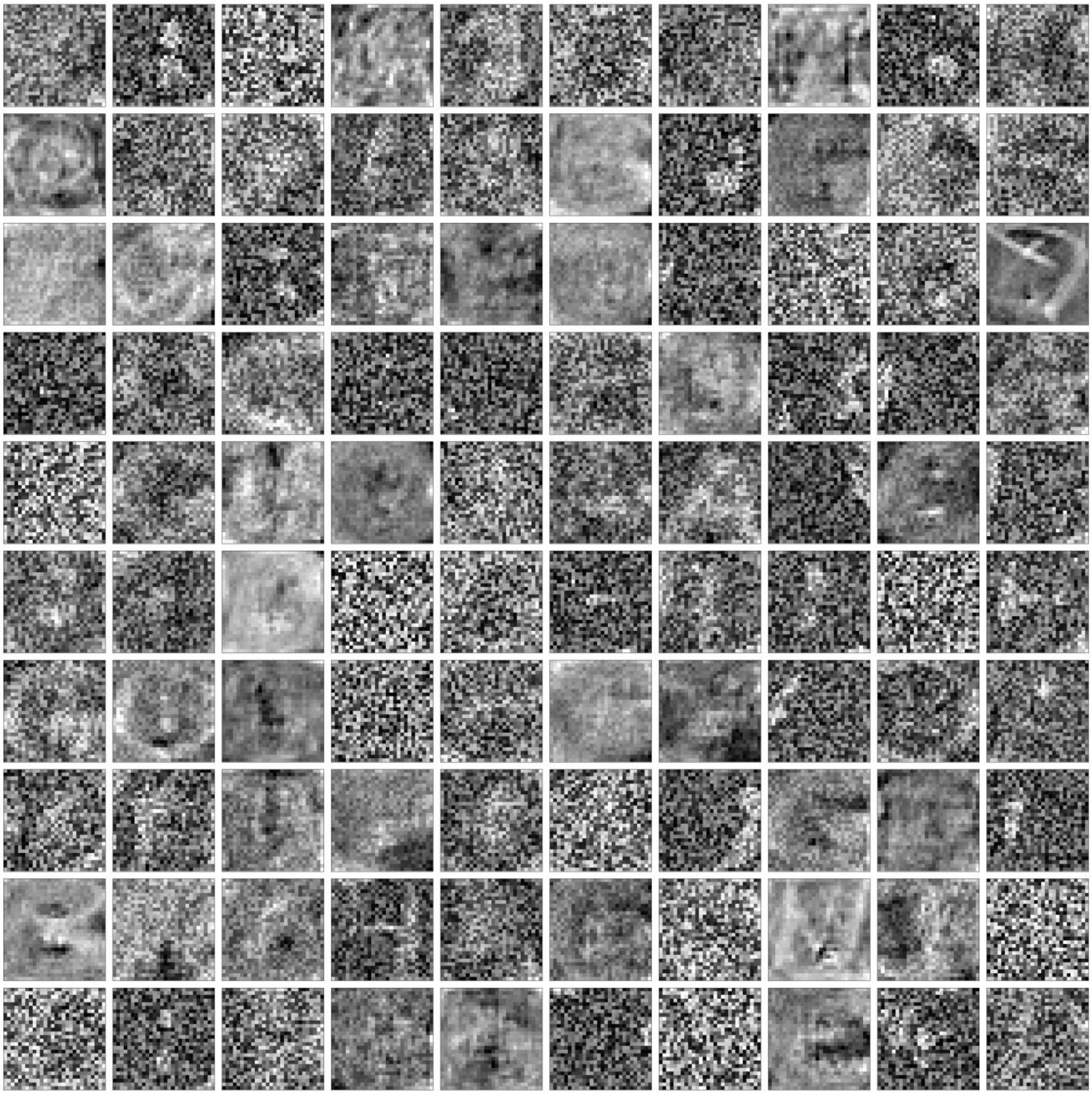




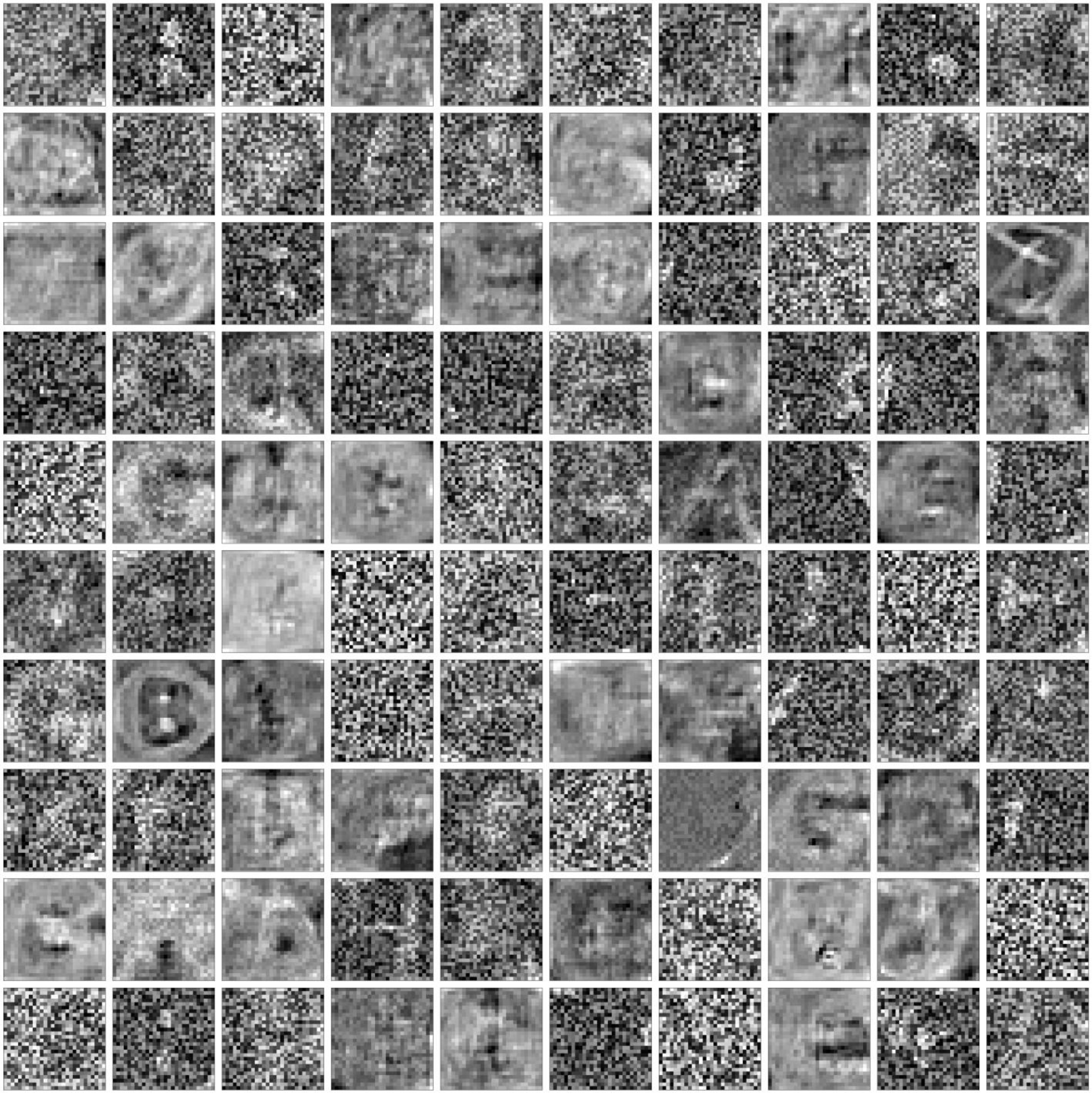
1.3.2 – Visualization

100 of the neurons are displayed in the images before at 25% and 100% completion of training for non-dropout and dropout cases. We can see that from 25% to 100% the images start to become clearer and you can start to make out letters for some neurons. We also see that the neurons into the dropout case are in general fuzzier than that of the non-dropout case.

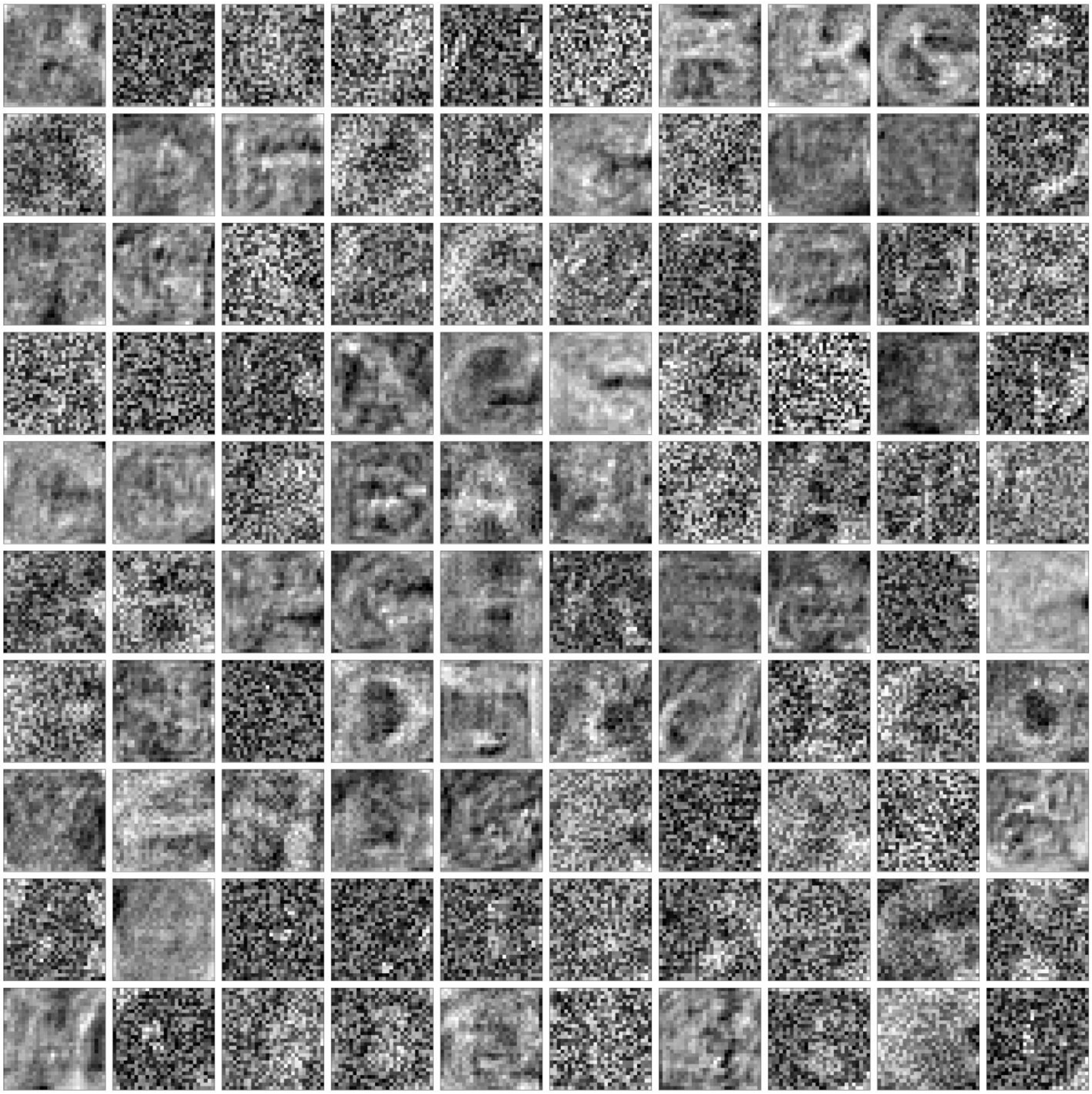
25% complete without dropout



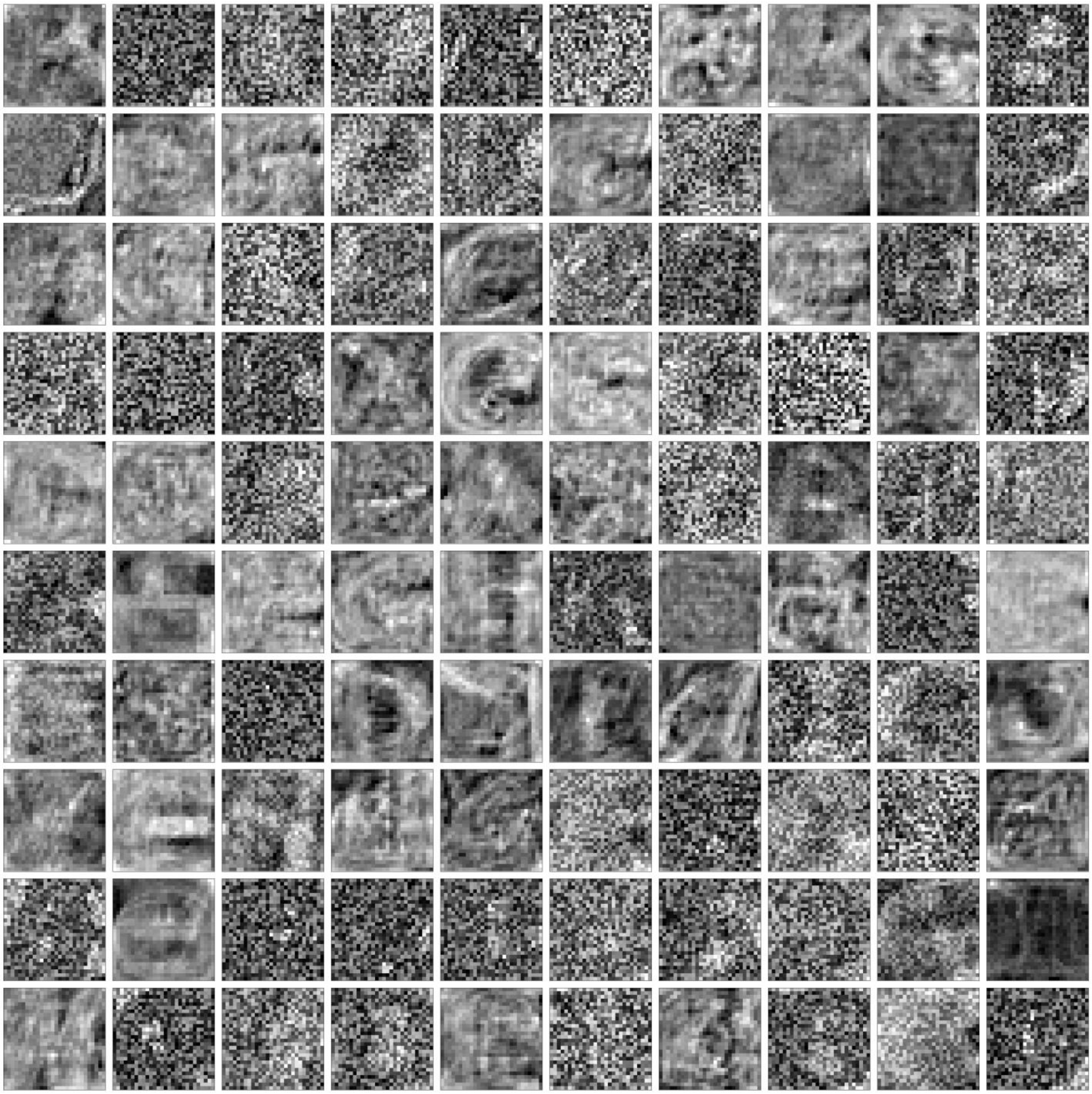
100% complete without dropout



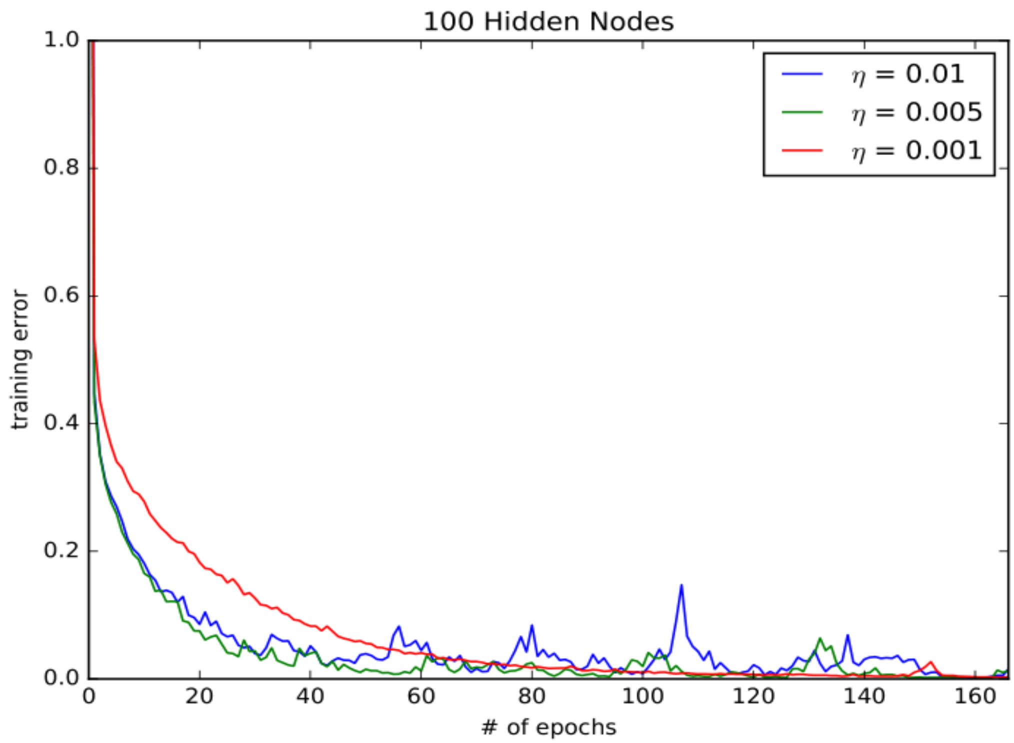
25% complete with dropout

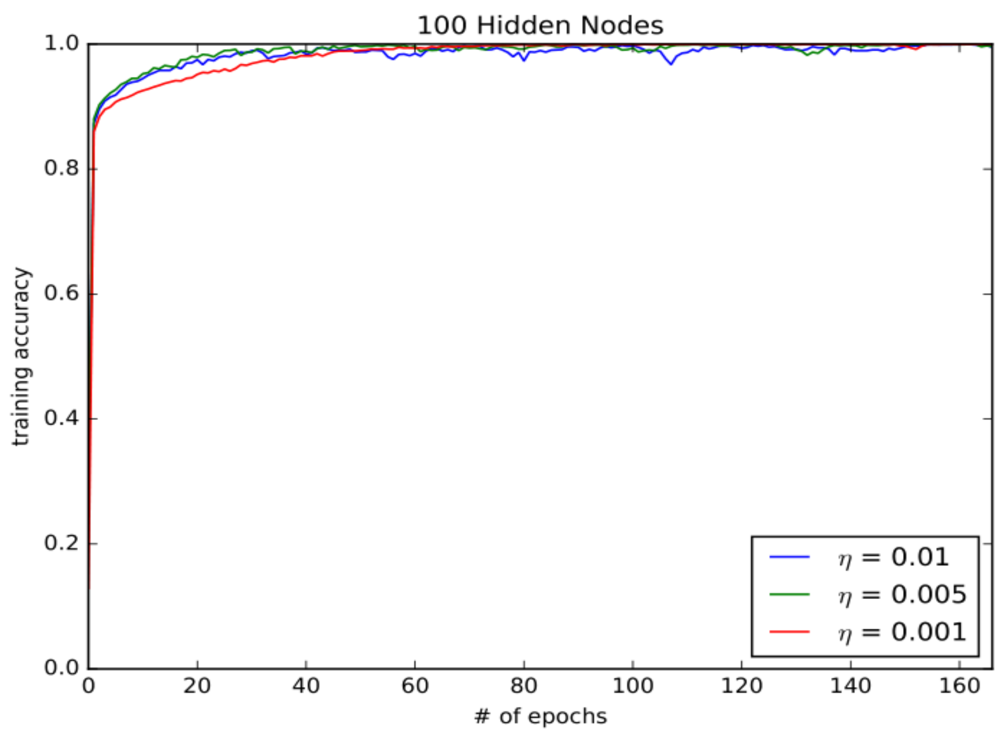


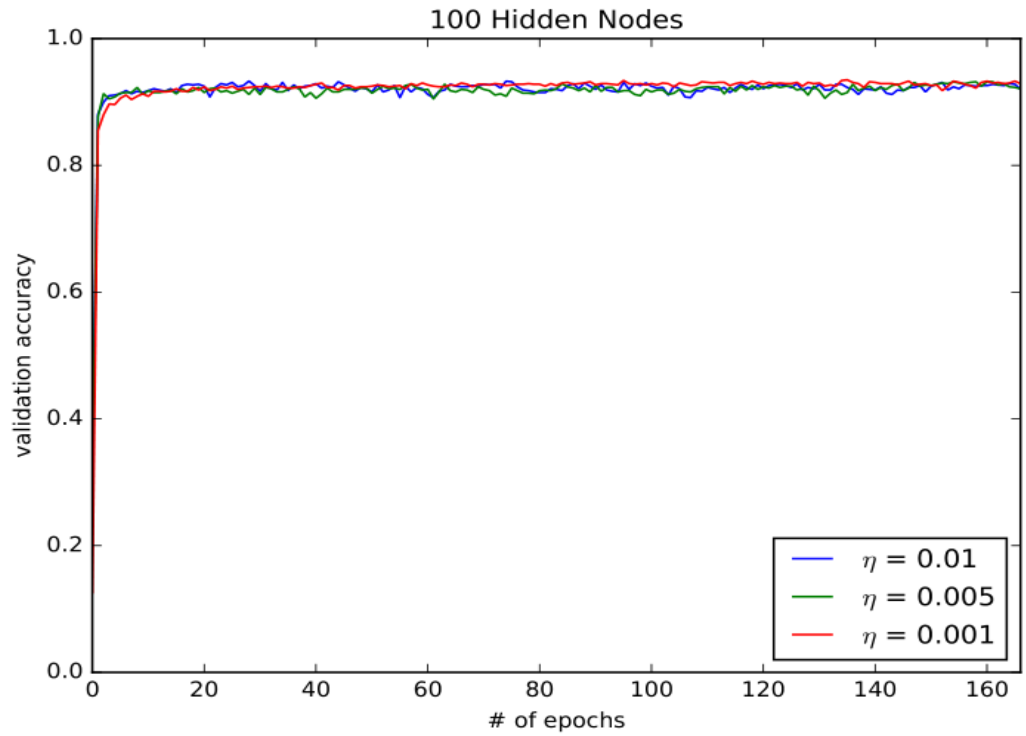
100% complete with dropout

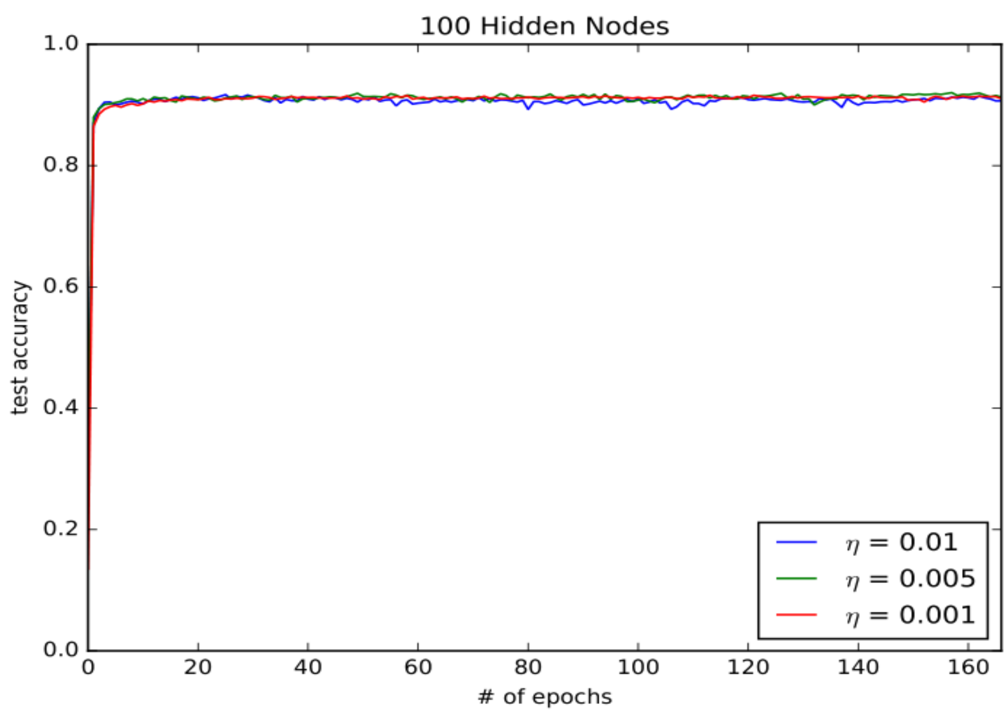


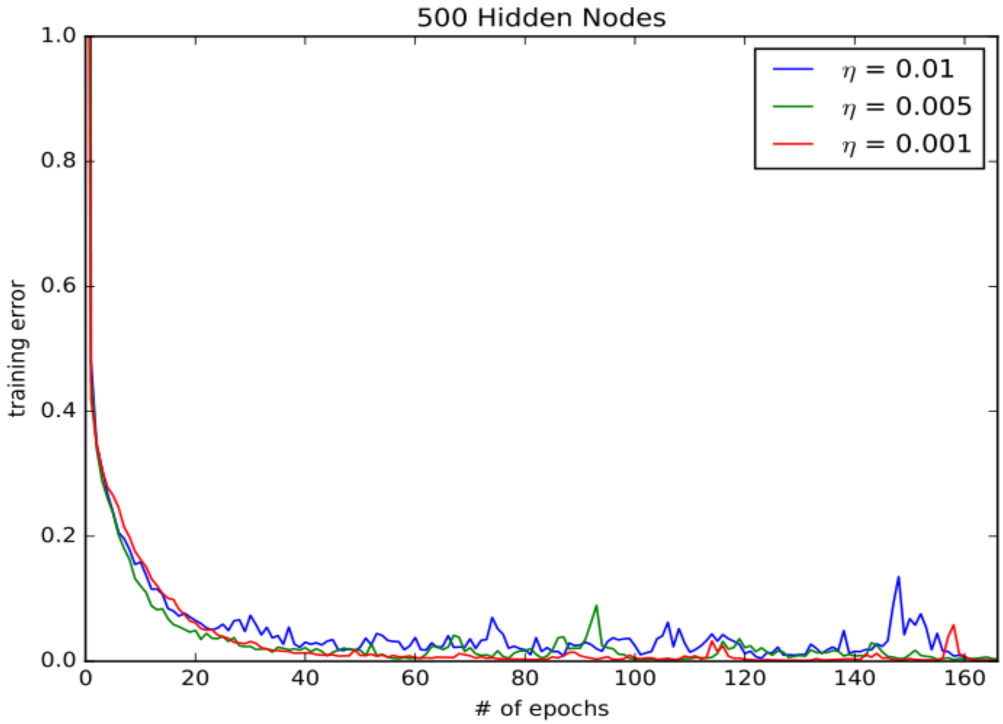
GRAPHS for 1.2.1

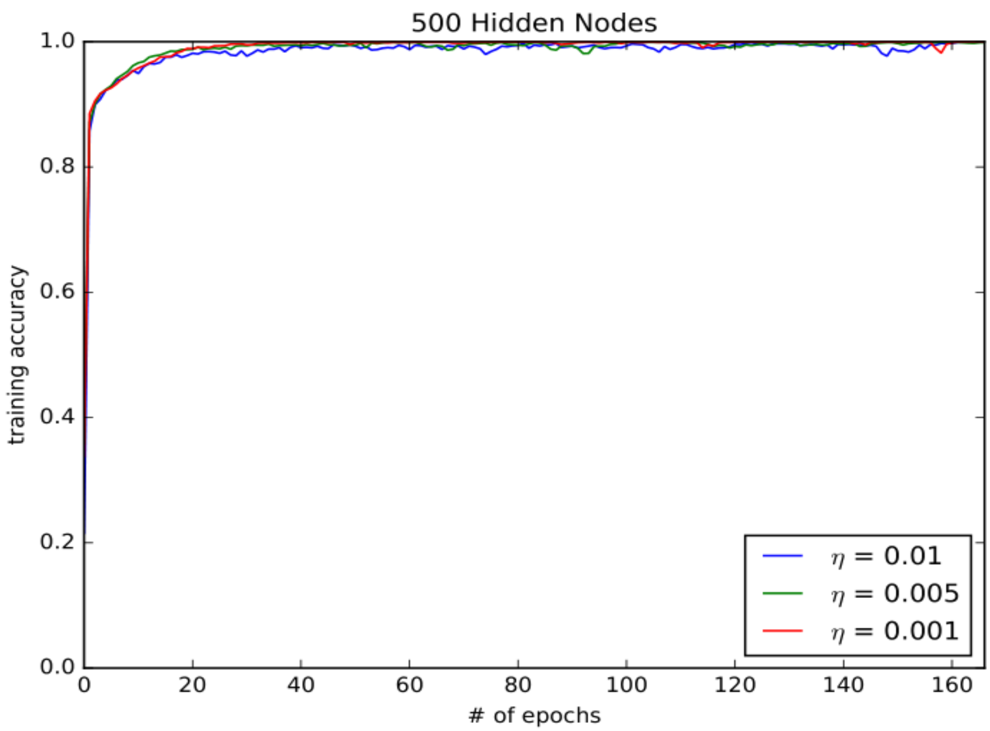


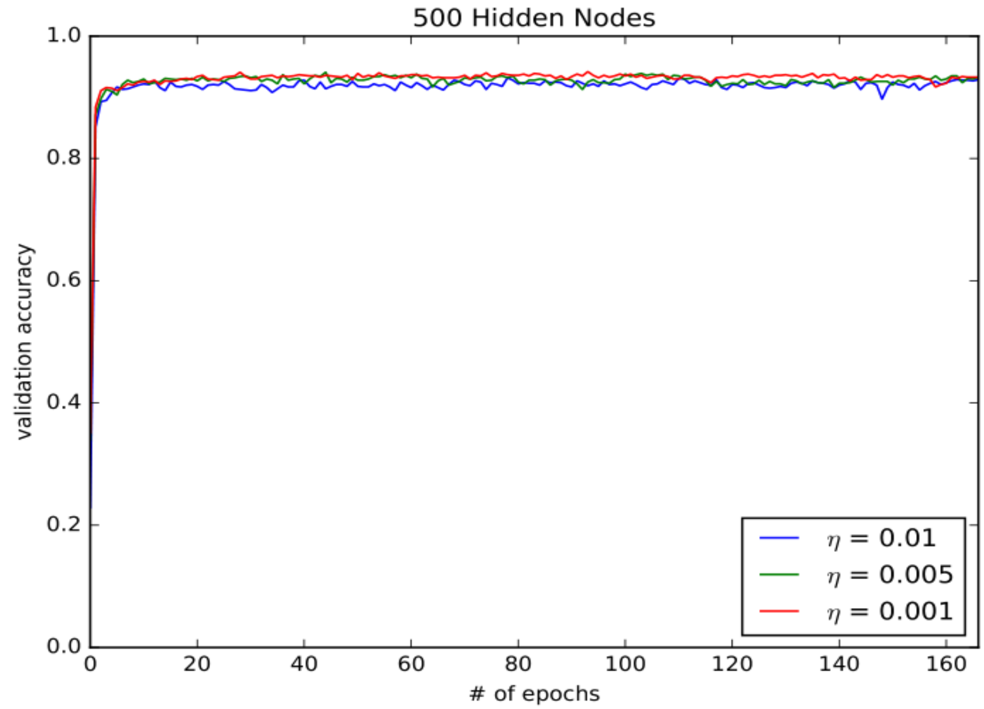


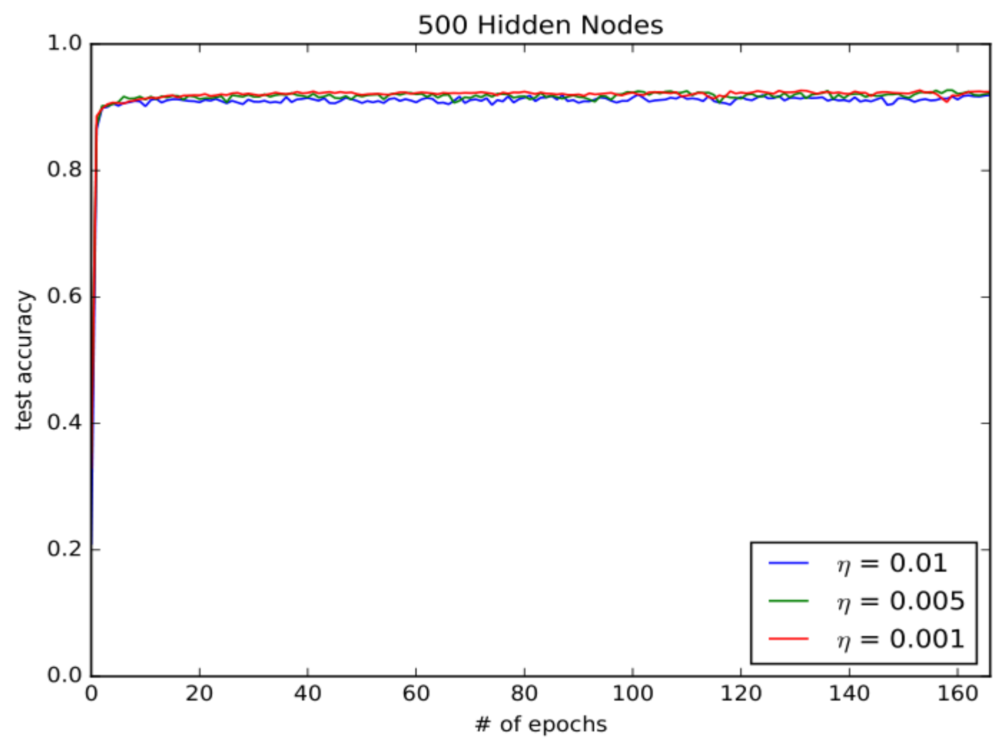

100 Hidden Nodes
validation accuracy
# of epochs
η = 0.01
η = 0.005
η = 0.001
0.0
0.2
0.4
0.6
0.8
1.0
0
20
40
60
80
100
120
140
160

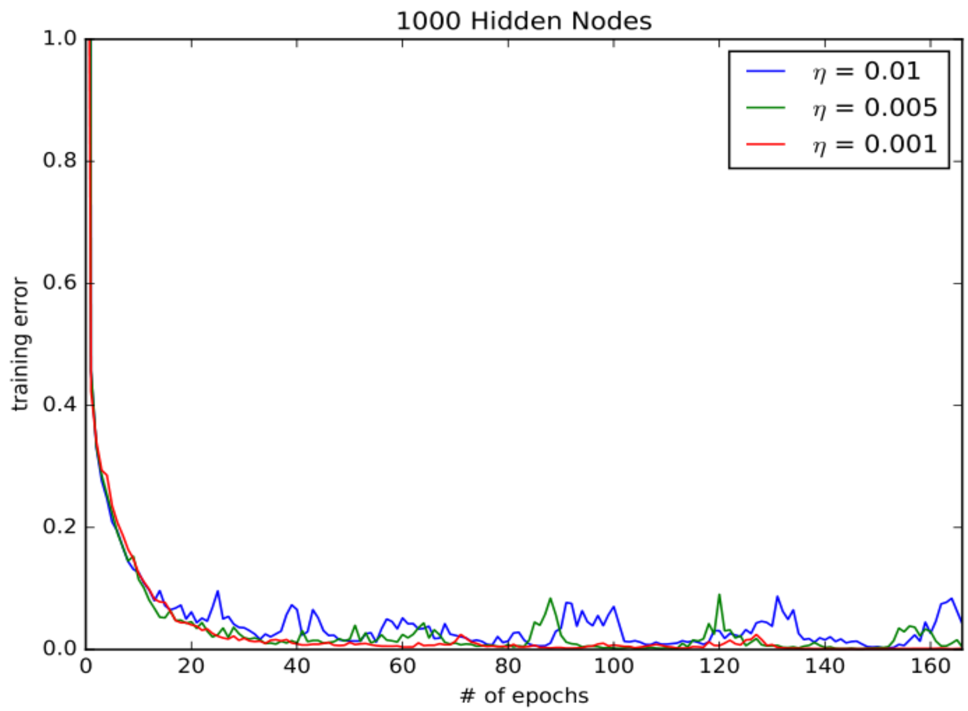

100 Hidden Nodes
test accuracy
# of epochs
η = 0.01
η = 0.005
η = 0.001
0.0
0.2
0.4
0.6
0.8
1.0
0
20
40
60
80
100
120
140
160

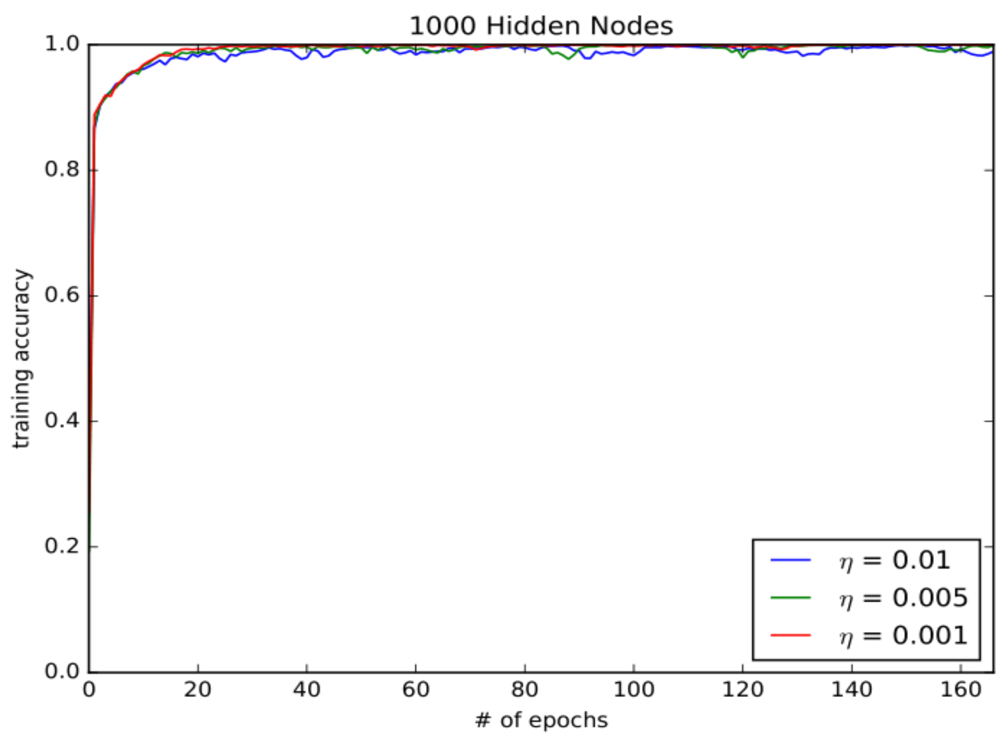

500 Hidden Nodes
training error
# of epochs
η = 0.01
η = 0.005
η = 0.001

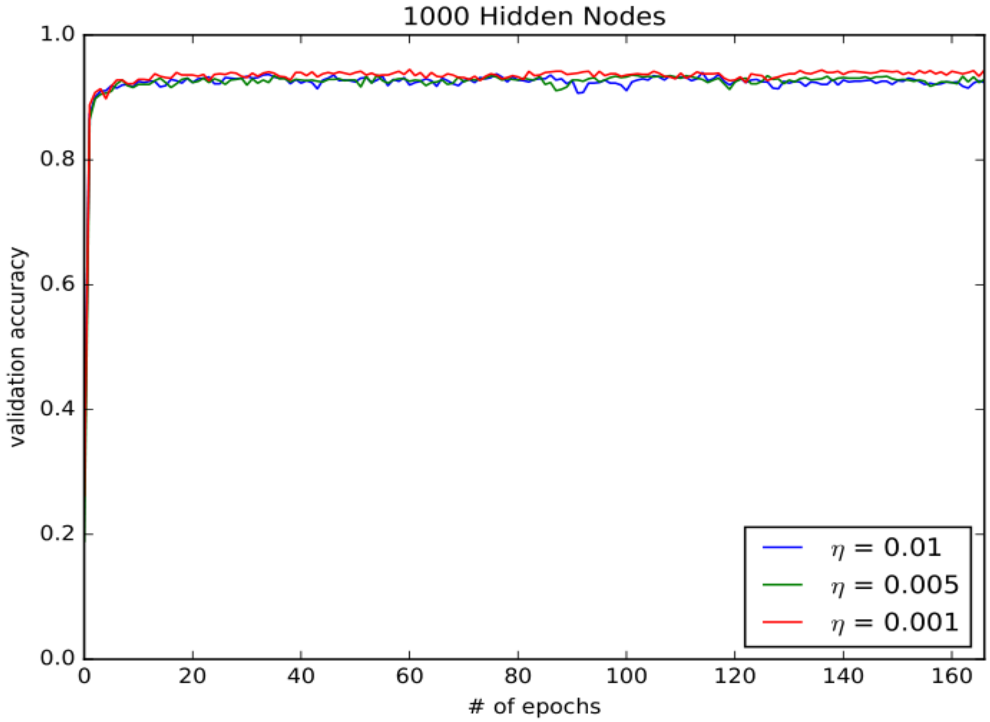

500 Hidden Nodes
training accuracy
# of epochs
η = 0.01
η = 0.005
η = 0.001

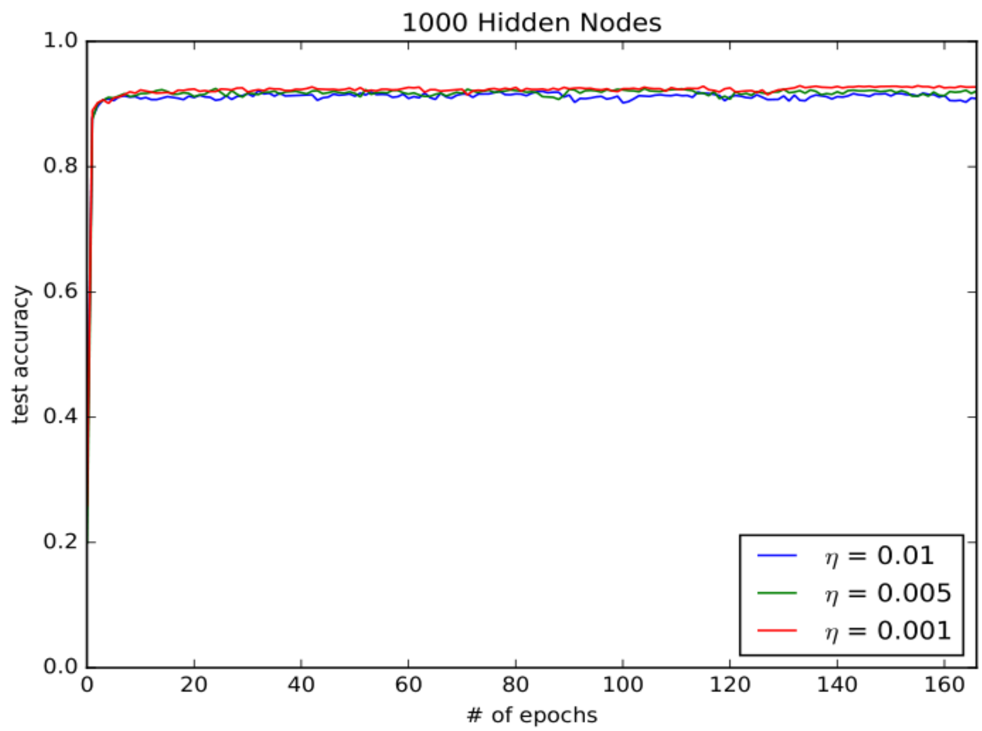

500 Hidden Nodes
validation accuracy
# of epochs
η = 0.01
η = 0.005
η = 0.001
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
120
140
160


500 Hidden Nodes
test accuracy
# of epochs
η = 0.01
η = 0.005
η = 0.001
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
120
140
160


1000 Hidden Nodes
training error
# of epochs
η = 0.01
η = 0.005
η = 0.001


1000 Hidden Nodes
training accuracy
# of epochs
η = 0.01
η = 0.005
η = 0.001


1000 Hidden Nodes
1.0
0.8
0.6
0.4
0.2
0.0
validation accuracy
0
20
40
60
80
100
120
140
160
# of epochs
η = 0.01
η = 0.005
η = 0.001


1000 Hidden Nodes
1.0
0.8
0.6
0.4
0.2
0.0
test accuracy
0
20
40
60
80
100
120
140
160
# of epochs
η = 0.01
η = 0.005
η = 0.001

Python Codes

neural_networks.py

```
import tensorflow as tf
import numpy as np

# Data loader for notMNIST dataset
def load_notmnist_data():
    with np.load("notMNIST.npz") as data:
        Data, Target = data["images"], data["labels"]
        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]
        t = np.zeros((trainTarget.shape[0], 10))
        t[np.arange(trainTarget.shape[0]), trainTarget] = 1
        trainTarget = t
        validData, validTarget = Data[15000:16000], Target[15000:16000]
        t = np.zeros((validTarget.shape[0], 10))
        t[np.arange(validTarget.shape[0]), validTarget] = 1
        validTarget = t
        testData, testTarget = Data[16000:], Target[16000:]
        t = np.zeros((testTarget.shape[0], 10))
        t[np.arange(testTarget.shape[0]), testTarget] = 1
        testTarget = t
        return (trainData.reshape(trainData.shape[0], -1), trainTarget, validData.reshape(validData.shape[0], -1), validTarget,
testData.reshape(testData.shape[0], -1), testTarget)

# Q1.1.1 layer-wise building block
def create_new_layer(input_tensor, num_hidden_units):
    '''
        @param input_tensor - outputs of the previous layer in the neural network, without the bias term.
        @param num_hidden_units - number of hidden units to use for this new layer
    '''
    # Create the new layer weight matrix using Xavier initialization
    input_dim = int(input_tensor.shape[-1])
    initializer = tf.contrib.layers.xavier_initializer()
    W_shape = [input_dim, num_hidden_units]
    W = tf.get_variable("Layer1_W", initializer=initializer(W_shape), dtype=tf.float32)
    # todo: zero initializer?
    b = tf.get_variable("Layer1_b", shape=[1, num_hidden_units], dtype=tf.float32)

    # MatMul the extended input tensor by the new weight matrix and add the biases
    output_tensor = tf.matmul(input_tensor, W) + b

    # Return this operation
    return output_tensor

# Q1.1.2 learning
def learning():
    xTrain, yTrain, xValid, yValid, xTest, yTest = load_notmnist_data()
```

```
with tf.Graph().as_default():
    num_hidden_units = 1000
    decay = 0
    B = 500
    learning_rates = [0.01, 0.005, 0.001]
    iters = 5000

    num_iters_per_epoch = len(xTrain)//B # number of iterations we have to do for one epoch
    print("Num epochs = ",iters/num_iters_per_epoch)

    # hyperparameters
    learning_rate = tf.placeholder(dtype=tf.float32, name="learning-rate")

    # Get Data
    xTrainTensor = tf.constant(xTrain, dtype=tf.float32, name="X-Training")
    yTrainTensor = tf.constant(yTrain, dtype=tf.float32, name="Y-Training")
    xTestTensor = tf.constant(xTest, dtype=tf.float32, name="X-Test")
    yTestTensor = tf.constant(yTest, dtype=tf.float32, name="Y-Test")
    xValidTensor = tf.constant(xValid, dtype=tf.float32, name="X-Validation")
    yValidTensor = tf.constant(yValid, dtype=tf.float32, name="Y-Validation")

    Xslice, yslice = tf.train.slice_input_producer([xTrainTensor, yTrainTensor], num_epochs=None)

    Xbatch, ybatch = tf.train.batch([Xslice, yslice], batch_size = B)

    with tf.variable_scope("default") as scope:
        # Create neural network layers for training
        trainb_batchOutput = create_new_layer(Xbatch, num_hidden_units)
        trainb_activatedOutput = tf.nn.relu(trainb_batchOutput)

        scope.reuse_variables()
        layer1_w = tf.get_variable("Layer1_W", shape=[784, num_hidden_units], dtype=tf.float32)
        layer1_b = tf.get_variable("Layer1_b", shape=[1, num_hidden_units], dtype=tf.float32)

        train_output = tf.matmul(xTrainTensor, layer1_w) + layer1_b
        train_activatedOutput = tf.nn.relu(train_output)

        valid_output = tf.matmul(xValidTensor, layer1_w) + layer1_b
        valid_activatedOutput = tf.nn.relu(valid_output)

        test_output = tf.matmul(xTestTensor, layer1_w) + layer1_b
        test_activatedOutput = tf.nn.relu(test_output)

        outputWeights_size = [int(trainb_activatedOutput.shape[-1]), 10] # We want a [1,10] tensor to get probabilities for
each class
        outputWeights = tf.Variable(tf.contrib.layers.xavier_initializer()(outputWeights_size), name="Output_W")
        outputBias = tf.Variable(0, dtype=tf.float32, name="Output_Bias")

        trainb_y_pred = tf.sigmoid(tf.matmul(trainb_activatedOutput, outputWeights) + outputBias)
        train_y_pred = tf.sigmoid(tf.matmul(train_activatedOutput, outputWeights) + outputBias)
        valid_y_pred = tf.sigmoid(tf.matmul(valid_activatedOutput, outputWeights) + outputBias)
        test_y_pred = tf.sigmoid(tf.matmul(test_activatedOutput, outputWeights) + outputBias)
        trainb_softmaxLoss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=trainb_y_pred,
labels=ybatch)) + decay * tf.nn.l2_loss(layer1_w)
```

```
        train_softmaxLoss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=train_y_pred, 
labels=yTrainTensor)) + decay * tf.nn.l2_loss(layer1_w)

        train_accuracy = tf.count_nonzero(tf.equal(tf.argmax(train_y_pred, 1), tf.argmax(yTrainTensor, 1))) / 
yTrainTensor.shape[0]
        valid_accuracy = tf.count_nonzero(tf.equal(tf.argmax(valid_y_pred, 1), tf.argmax(yValidTensor, 1))) / 
yValidTensor.shape[0]
        test_accuracy = tf.count_nonzero(tf.equal(tf.argmax(test_y_pred, 1), tf.argmax(yTestTensor, 1))) / 
yTestTensor.shape[0]

    # optimizer function
    optimizer = tf.train.AdamOptimizer(learning_rate).minimize(trainb_softmaxLoss)

    # TODO run a lot of iterations, plot loss vs epochs and classification error vs epochs
    for r in learning_rates:
        loss_amounts = []
        train_accs = []
        test_accs = []
        valid_accs = []
        with tf.Session() as sess:
            coord = tf.train.Coordinator()
            threads = tf.train.start_queue_runners(sess=sess, coord=coord)
            sess.run(tf.global_variables_initializer())
            sess.run(tf.local_variables_initializer())
            for i in range(iters):
                sess.run([optimizer], feed_dict={learning_rate: r})
                if (i % num_iters_per_epoch == 0):
                    t_loss, t_acc, v_acc, test_acc = sess.run([train_softmaxLoss, train_accuracy, valid_accuracy, test_accuracy])
                    print("Epoch: {}, Training Loss: {}, Accuracies: [{}, {}, {}]".format(i//num_iters_per_epoch, t_loss, t_acc, v_acc, 
test_acc))
                    loss_amounts.append(t_loss)
                    train_accs.append(t_acc)
                    test_accs.append(test_acc)
                    valid_accs.append(v_acc)
            np.save("1.1.2_r{}_loss".format(r), loss_amounts)
            np.save("1.1.2_r{}_train_acc".format(r), train_accs)
            np.save("1.1.2_r{}_test_acc".format(r), test_accs)
            np.save("1.1.2_r{}_valid_acc".format(r), valid_accs)
learning()
```

effect_of_hyperparameters.py

```
import numpy as np
import tensorflow as tf

def load_notMNIST():
    with np.load("notMNIST.npz") as data:
        Data, Target = data["images"], data["labels"]
        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]
        t = np.zeros((trainTarget.shape[0], 10))
        t[np.arange(trainTarget.shape[0]), trainTarget] = 1
        trainTarget = t
        validData, validTarget = Data[15000:16000], Target[15000:16000]
        t = np.zeros((validTarget.shape[0], 10))
        t[np.arange(validTarget.shape[0]), validTarget] = 1
        validTarget = t
        testData, testTarget = Data[16000:], Target[16000:]
        t = np.zeros((testTarget.shape[0], 10))
        t[np.arange(testTarget.shape[0]), testTarget] = 1
        testTarget = t
        return (trainData.reshape(trainData.shape[0], -1), trainTarget, validData.reshape(validData.shape[0], -1),
validTarget, testData.reshape(testData.shape[0], -1), testTarget)

def create_new_layer(input_tensor, num_hidden_units):
    '''
        @param input_tensor - outputs of the previous layer in the neural network, without the bias term.
        @param num_hidden_units - number of hidden units to use for this new layer
    '''
    # Create the new layer weight matrix using Xavier initialization
    input_dim = int(input_tensor.shape[-1])
    initializer = tf.contrib.layers.xavier_initializer()
    W_shape = [input_dim, num_hidden_units]
    W = tf.get_variable("W", initializer=initializer(W_shape), dtype=tf.float32)
    # todo: zero initializer?
    b = tf.get_variable("b", shape=[1, num_hidden_units], dtype=tf.float32)

    # MatMul the extended input tensor by the new weight matrix and add the biases
    output_tensor = tf.matmul(input_tensor, W) + b

    # Return this operation
    return output_tensor

def number_of_hidden_units():
    # Constants
    B = 500
    iters = 5000
    learning_rates = [0.01, 0.005, 0.001]
    hidden_units = [100,500,1000]
    output_data = [[],[],[]]

    # Load data
```

```
    (trainData, trainTarget, validData, validTarget,
        testData, testTarget) = load_notMNIST()

    # Precalculations
    num_iters_per_epoch = len(trainData)//B # number of iterations we have to do for one epoch
    print("Num epochs = ",iters/num_iters_per_epoch)
    inds = np.arange(trainData.shape[0])

    # Set place-holders & variables
    X = tf.placeholder(tf.float32, shape=(None, trainData.shape[-1]), name='X')
    Y = tf.placeholder(tf.float32, shape=(None, 10), name='Y')
    learning_rate = tf.placeholder(tf.float32, name='learning-rate')

    for h in range(0, len(hidden_units)):
        for lr in range(len(learning_rates)):
            # Build graph
            with tf.variable_scope("layer1_"+str(hidden_units[h])+"_"+str(lr), reuse=tf.AUTO_REUSE):
                s_1 = create_new_layer(X, hidden_units[h])
            x_1 = tf.nn.relu(s_1)
            with tf.variable_scope("layer2_"+str(hidden_units[h])+"_"+str(lr), reuse=tf.AUTO_REUSE):
                s_2 = create_new_layer(x_1, 10)
            x_2 = tf.nn.softmax(s_2)

            # Calculate loss & accuracy
            loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=s_2, labels=Y))
            accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(x_2, 1), tf.argmax(Y, 1)), tf.float32))

            print("Number of hidden units", hidden_units[h])

            with tf.Session() as sess:
                with tf.variable_scope("default", reuse=tf.AUTO_REUSE):
                    optimizer = tf.train.AdamOptimizer(learning_rate).minimize(loss)
                    coord = tf.train.Coordinator()
                    threads = tf.train.start_queue_runners(sess=sess, coord=coord)
                    sess.run(tf.global_variables_initializer())
                    sess.run(tf.local_variables_initializer())
                    print("Learning rate = ",learning_rates[lr])
                    temp_output = []
                    for i in range(iters):
                        if (i % num_iters_per_epoch == 0):
                            np.random.shuffle(inds)
                        sess.run([optimizer], feed_dict={learning_rate: learning_rates[lr],
                                X: trainData[inds[B*(i%num_iters_per_epoch):B*((i+1)%num_iters_per_epoch)]],
                                Y: trainTarget[inds[B*(i%num_iters_per_epoch):B*((i+1)%num_iters_per_epoch)]]})
                        if (i % num_iters_per_epoch == 0):
                            t_loss, t_acc = sess.run([loss, accuracy], feed_dict={X: trainData, Y: trainTarget})
                            v_loss, v_acc = sess.run([loss, accuracy], feed_dict={X: validData, Y: validTarget})
                            test_loss, test_acc = sess.run([loss, accuracy], feed_dict={X: testData, Y: testTarget})
                            print("Epoch: {}, Training Loss: {}, Accuracies: [{}, {}, {}]".format(i//num_iters_per_epoch, t_loss,
t_acc, v_acc, test_acc))
                            temp_output.append([t_loss, t_acc, v_acc, test_acc])
                    output_data[h].append(temp_output)

    np.save('Q1-2-1.npy', output_data)
    return output_data
```

```
def number_of_layers():
    # Constants
    B = 250
    iters = 5000
    learning_rates = [0.01, 0.005, 0.001, 0.0005, 0.0001]
    hidden_units = [500]
    output_data = [[]]

    # Load data
    (trainData, trainTarget, validData, validTarget,
        testData, testTarget) = load_notMNIST()

    # Precalculations
    num_iters_per_epoch = len(trainData)//B # number of iterations we have to do for one epoch
    print("Num epochs = ",iters/num_iters_per_epoch)
    inds = np.arange(trainData.shape[0])

    # Set place-holders & variables
    X = tf.placeholder(tf.float32, shape=(None, trainData.shape[-1]), name='X')
    Y = tf.placeholder(tf.float32, shape=(None, 10), name='Y')
    learning_rate = tf.placeholder(tf.float32, name='learning-rate')

    for h in range(0, len(hidden_units)):
        for lr in range(len(learning_rates)):
            # Build graph
            with tf.variable_scope("layer1_"+str(hidden_units[h])+"_"+str(lr), reuse=tf.AUTO_REUSE):
                s_1 = create_new_layer(X, hidden_units[h])
            x_1 = tf.nn.relu(s_1)
            with tf.variable_scope("layer2_"+str(hidden_units[h])+"_"+str(lr), reuse=tf.AUTO_REUSE):
                s_2 = create_new_layer(x_1, hidden_units[h])
            x_2 = tf.nn.softmax(s_2)
            with tf.variable_scope("layer3_"+str(hidden_units[h])+"_"+str(lr), reuse=tf.AUTO_REUSE):
                s_3 = create_new_layer(x_2, 10)
            x_3 = tf.nn.softmax(s_3)

            # Calculate loss & accuracy
            loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=s_3, labels=Y))
            accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(x_3, 1), tf.argmax(Y, 1)), tf.float32))

            print("Number of hidden layers: 2, Number of hidden units", hidden_units[h])

            with tf.Session() as sess:
                with tf.variable_scope("default", reuse=tf.AUTO_REUSE):
                    optimizer = tf.train.AdamOptimizer(learning_rate).minimize(loss)
                    coord = tf.train.Coordinator()
                    threads = tf.train.start_queue_runners(sess=sess, coord=coord)
                    sess.run(tf.global_variables_initializer())
                    sess.run(tf.local_variables_initializer())
                    print("Learning rate = ",learning_rates[lr])
                    temp_output = []
                    for i in range(iters):
                        if (i % num_iters_per_epoch == 0):
                            np.random.shuffle(inds)
                        sess.run([optimizer], feed_dict={learning_rate: learning_rates[lr],
```

```
                    X: trainData[inds[B*(i%num_iters_per_epoch):B*((i+1)%num_iters_per_epoch)]],
                    Y: trainTarget[inds[B*(i%num_iters_per_epoch):B*((i+1)%num_iters_per_epoch)]]})
                if (i % num_iters_per_epoch == 0):
                    t_loss, t_acc = sess.run([loss, accuracy], feed_dict={X: trainData, Y: trainTarget})
                    v_loss, v_acc = sess.run([loss, accuracy], feed_dict={X: validData, Y: validTarget})
                    test_loss, test_acc = sess.run([loss, accuracy], feed_dict={X: testData, Y: testTarget})
                    print("Epoch: {}, Training Loss: {}, Accuracies: [{}, {}, {}]".format(i//num_iters_per_epoch, t_loss,
t_acc, v_acc, test_acc))
                    temp_output.append([t_loss, t_acc, v_acc, test_acc])
            output_data[h].append(temp_output)

    np.save('Q1-2-2.npy', output_data)
    return output_data

#output = number_of_hidden_units()
output = number_of_layers()
```

regularization_and_visualization.py

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

def dropout(x, is_training, p):
  return tf.cond(is_training, lambda: tf.nn.dropout(x, p, name='dropout'), lambda: tf.identity(x))

# Data loader for notMNIST dataset
def load_notmnist_data():
  with np.load("notMNIST.npz") as data:
    Data, Target = data["images"], data["labels"]
    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]
    t = np.zeros((trainTarget.shape[0], 10))
    t[np.arange(trainTarget.shape[0]), trainTarget] = 1
    trainTarget = t
    validData, validTarget = Data[15000:16000], Target[15000:16000]
    t = np.zeros((validTarget.shape[0], 10))
    t[np.arange(validTarget.shape[0]), validTarget] = 1
    validTarget = t
    testData, testTarget = Data[16000:], Target[16000:]
    t = np.zeros((testTarget.shape[0], 10))
    t[np.arange(testTarget.shape[0]), testTarget] = 1
    testTarget = t
    return (trainData.reshape(trainData.shape[0], -1), trainTarget, validData.reshape(validData.shape[0], -1),
validTarget, testData.reshape(testData.shape[0], -1), testTarget)

def FCN(x, depth, name, use_dropout=False, is_training=tf.constant(False), use_relu=False):
  W = tf.get_variable(name=name + "_W", shape=(x.shape[1], depth), dtype=tf.float64)
  b = tf.get_variable(name=name+ "_b", shape=(depth,), dtype=tf.float64, initializer=tf.zeros_initializer)
  if use_dropout:
    if use_relu:
      return dropout(tf.nn.relu(tf.matmul(x, W) + b), is_training, 0.5)
    else:
      return dropout(tf.matmul(x, W) + b, is_training, 0.5)
  else:
    if use_relu:
      return tf.nn.relu(tf.matmul(x, W) + b)
    else:
      tf.matmul(x, W) + b

def build_network(input_node, is_training_t):

  #can be changed for 2-layer networks
  num_hidden_units = 1000
  L1_out = FCN(input_node[0], num_hidden_units, name='Layer_1', use_dropout=True,
is_training=is_training_t,
         use_relu=True)
  W = tf.get_variable(name="output_W", shape=(L1_out.shape[1], 10), dtype=tf.float64)
  b = tf.get_variable(name="output_b", shape=(10,), dtype=tf.float64, initializer=tf.zeros_initializer)
```

```
    #for multiple nerual networks
    # L1_out = FCN(input_node[0], num_hidden_units, name='Layer_1', use_dropout=True,
is_training=is_training_t, use_relu=True)
    # L2_out = FCN(L1_out, num_hidden_units, name='Layer_2', use_dropout=True, is_training=is_training_t,
    #           use_relu=True)
    # L3_out = FCN(L2_out, num_hidden_units, name='Layer_3', use_dropout=True, is_training=is_training_t,
    #           use_relu=True)
    # L4_out = FCN(L3_out, num_hidden_units, name='Layer_4', use_dropout=True, is_training=is_training_t,
    #           use_relu=True)
    # W = tf.get_variable(name="output_W", shape=(L1_out.shape[1], 10), dtype=tf.float64)
    # b = tf.get_variable(name="output_b", shape=(10,), dtype=tf.float64, initializer=tf.zeros_initializer)

    # y_pred_raw = tf.matmul(L4_out, W) + b

    y_pred_raw = tf.matmul(L1_out, W) + b
    return y_pred_raw

def learning():
    xTrain, yTrain, xValid, yValid, xTest, yTest = load_notmnist_data()

    with tf.Graph().as_default():
        num_hidden_units = 1000
        decay = 0
        B = 500
        learning_rates = [0.01, 0.005, 0.001]
        iters = 5000
        max_num_epochs = (B*iters)//len(xTrain)
        if B*iters % len(xTrain):
            max_num_epochs += 1
        num_iters_per_epoch = len(xTrain) // B  # number of iterations we have to do for one epoch
        print("Num epochs = ", iters / num_iters_per_epoch)

        # hyperparameters
        learning_rate = tf.placeholder(dtype=tf.float64, name="learning-rate")
        is_training_t = tf.placeholder(dtype=tf.bool, name="is_training")

        base_iterator = tf.data.Iterator.from_structure((tf.float64, tf.float64), ((None, 784), (None, 10)))
        input_node = base_iterator.get_next()
        y_pred_raw = build_network(input_node, is_training_t)

        y_pred = tf.nn.softmax(y_pred_raw)
        CE_loss = tf.losses.softmax_cross_entropy(input_node[1], y_pred_raw)

        vars = tf.global_variables()
        l2s = []
        for var in vars:
            l2s.append(tf.nn.l2_loss(var))
        l2_loss = tf.reduce_sum(tf.stack(l2s, axis=0))
        total_loss = CE_loss + decay * l2_loss
        accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(y_pred, 1), tf.argmax(input_node[1], 1)), tf.float32))

        # optimizer function
        optimizer = tf.train.AdamOptimizer(learning_rate).minimize(total_loss)
```

```
X = tf.placeholder(dtype=tf.float64, name="X")
Y = tf.placeholder(dtype=tf.float64, name="Y")
Xdata = tf.data.Dataset.from_tensor_slices(X)
Ydata = tf.data.Dataset.from_tensor_slices(Y)
sample_dataset = tf.data.Dataset.zip((Xdata, Ydata))
batched_dataset = sample_dataset.batch(B)
# TODO run a lot of iterations, plot loss vs epochs and classification error vs epochs
accuracy_list = []
ce_list = []
check_points = [iters//4, iters//2, 3*iters//4, iters-1]
saver = tf.train.Saver(vars)
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    # initialize data input pippeline for training
    dataset_init = base_iterator.make_initializer(batched_dataset)
    for i in range(max_num_epochs):
        sess.run(dataset_init, feed_dict={X:xTrain, Y:yTrain})
        j = 0
        while True:
            try:
                sess.run([optimizer, ], feed_dict={learning_rate: 0.005, is_training_t: True})
                j += 1
                if i * num_iters_per_epoch + j in check_points:
                    saver.save(sess, '.\my_model', global_step=i)
            except tf.errors.OutOfRangeError:
                break

        # initialize data iterator for getting numbers to plot
        # on train
        sess.run(dataset_init, feed_dict={X: xTrain, Y: yTrain})
        this_acc = 0.0
        this_ce = 0.0
        j = 0
        while True:
            try:
                acc, ce = sess.run([accuracy, CE_loss], feed_dict={is_training_t: False})
                this_acc += acc
                this_ce += ce
                j += 1
            except tf.errors.OutOfRangeError:
                break
        train_acc = this_acc/j
        train_ce = this_ce/j
        # on val
        sess.run(dataset_init, feed_dict={X: xValid, Y: yValid})
        this_acc = 0.0
        this_ce = 0.0
        j = 0
        while True:
            try:
                acc, ce = sess.run([accuracy, CE_loss], feed_dict={is_training_t: False})
                this_acc += acc
                this_ce += ce
                j += 1
            except tf.errors.OutOfRangeError:
```

```
                break
            val_acc = this_acc / j
            val_ce = this_ce / j
            # on test
            sess.run(dataset_init, feed_dict={X: xTest, Y: yTest})
            this_acc = 0.0
            this_ce = 0.0
            j = 0
            while True:
                try:
                    acc, ce = sess.run([accuracy, CE_loss], feed_dict={is_training_t: False})
                    this_acc += acc
                    this_ce += ce
                    j += 1
                except tf.errors.OutOfRangeError:
                    break
            test_acc = this_acc / j
            test_ce = this_ce / j
            accuracy_list.append((train_acc, val_acc, test_acc))
            ce_list.append((train_ce, val_ce, test_ce))
            print("Epoch: {}, Training Loss: {}, Accuracies: [{}, {}, {}]".format(i,
                                                                train_ce, train_acc, val_acc,
                                                                test_acc))
    return accuracy_list, ce_list

def visualization(filepath, index=1):
    base_iterator = tf.data.Iterator.from_structure((tf.float64, tf.float64), ((None, 784), (None, 10)))
    input_node = base_iterator.get_next()
    is_training_t = tf.placeholder(dtype=tf.bool, name="is_training")
    _ = build_network(input_node, is_training_t)
    saver = tf.train.Saver(tf.global_variables())
    for var in tf.global_variables():
        if var.name == "Layer_1_W:0":
            l1_w = var
    with tf.Session() as sess:
        saver.restore(sess, filepath)
        layer1_W = sess.run(l1_w)
        target = layer1_W[:, index]
        plt.imshow(np.reshape(target, (28,28)))
        plt.show()

if __name__ == "__main__":
    #accs, ces = learning()
    #acc_array = np.array(accs)
    #x = np.arange(acc_array.shape[0])
    #plt.plot(x, acc_array)
    #plt.show()
    #ces_array = np.array(ces)
    #plt.plot(x, ces_array)
    #plt.show()

    (ac, ce) = learning()
```