Name: Supriyo Ghosh

Student ID: 215318728

Assignment: 2

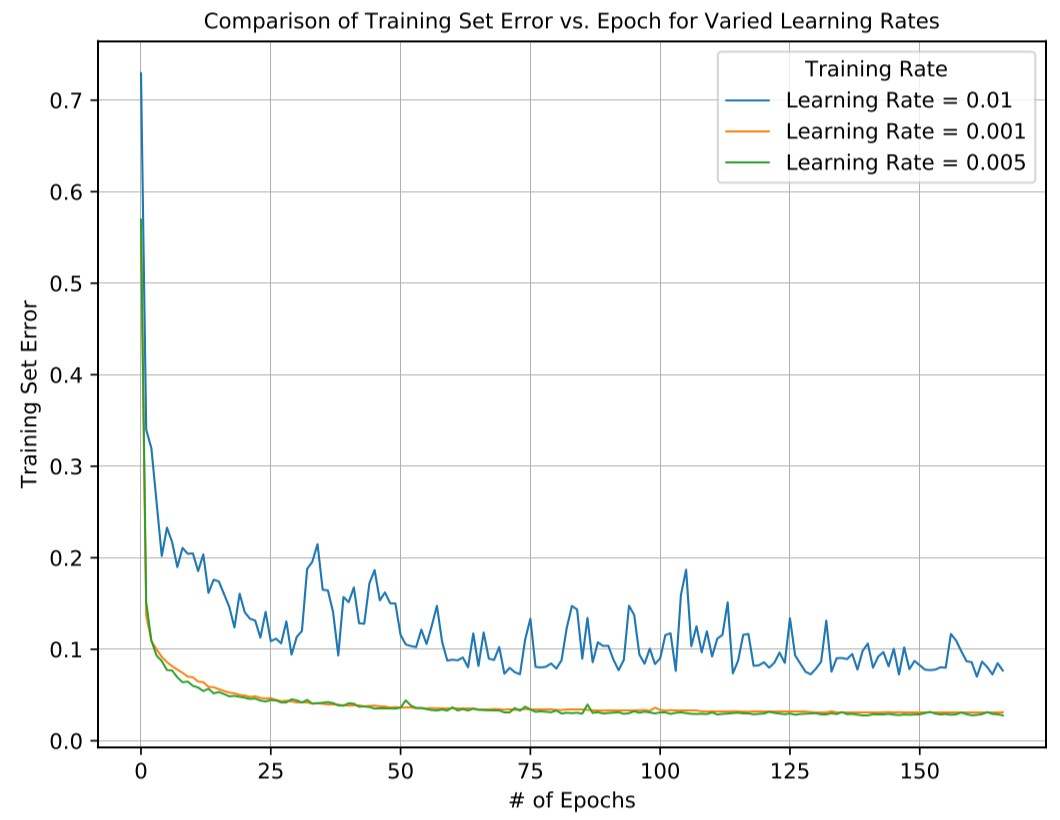
**1.1 – Feedforward fully connected neural networks**

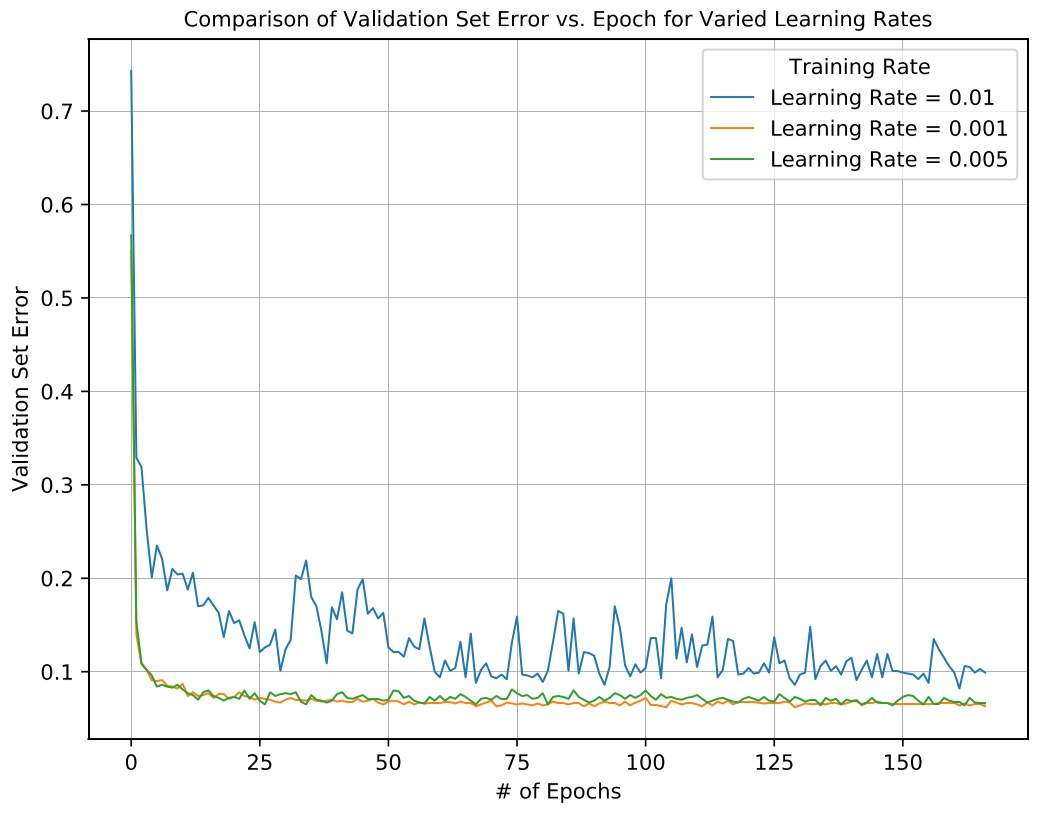
**1.1.1 – Layer-wise building block**

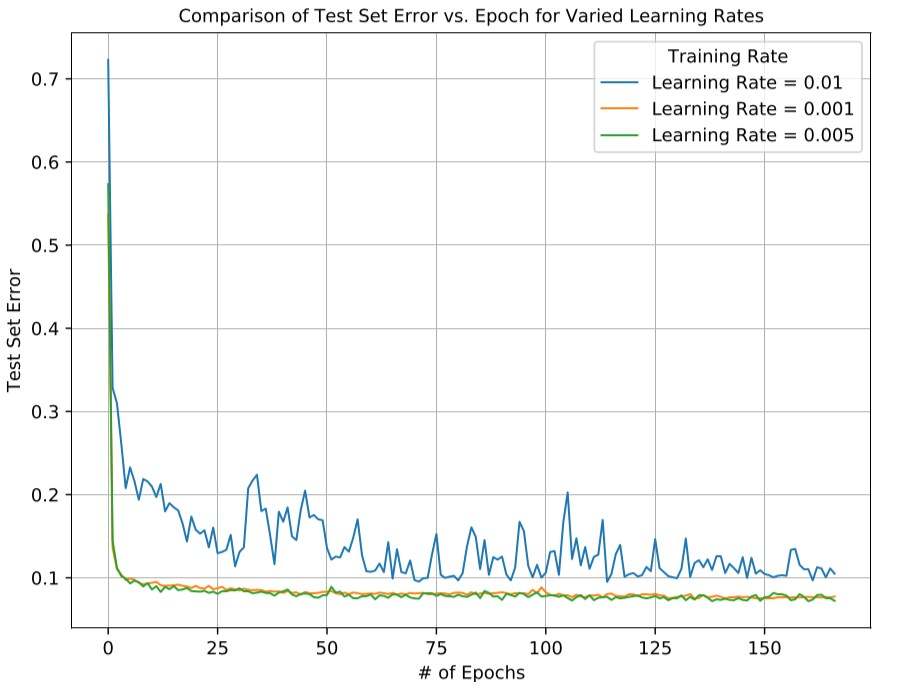
The *create\_new\_layer()*  function creates a new neural network layer from the input tensor *input\_tensor* with a specified number of hidden units. The input tensor represents the raw input vector without the bias padding. The weight matrix is initialized using Xavier initialization, and the bias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

**1.1.2 – Learning**

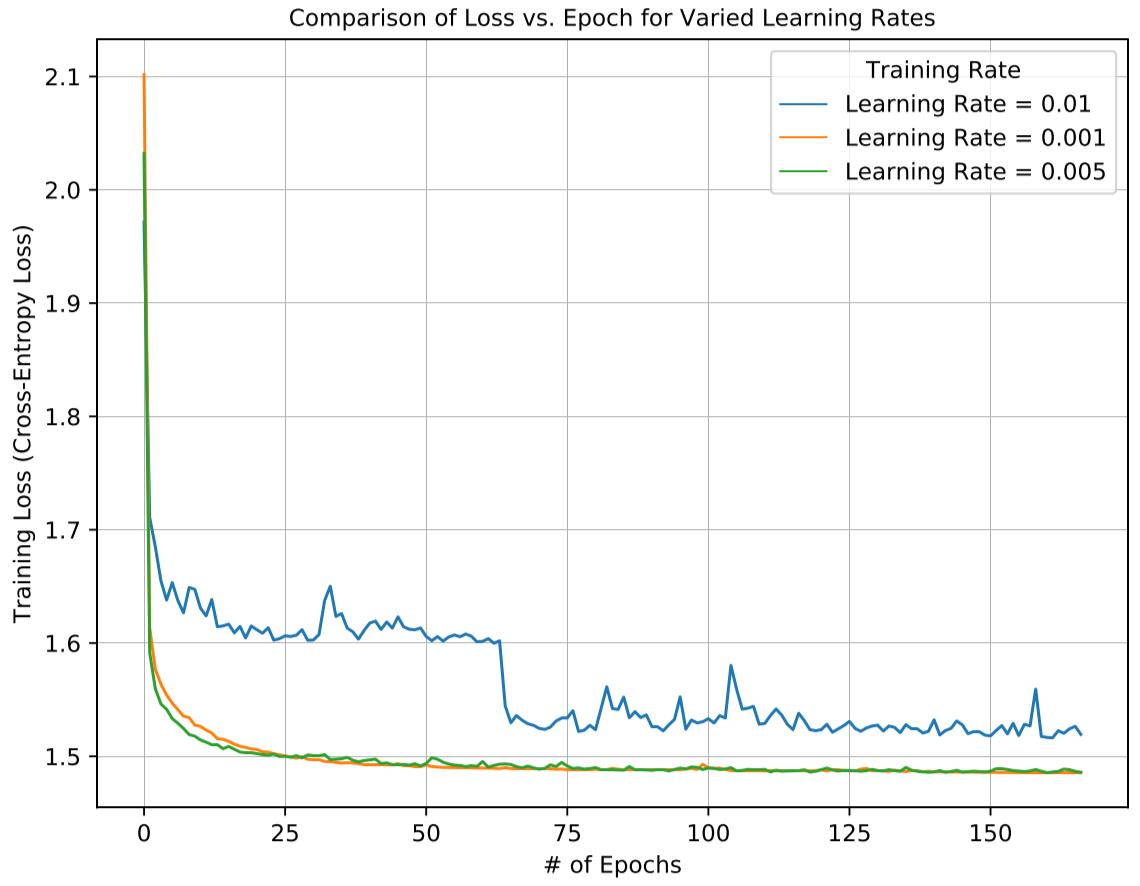
Using the above function, a simple one-layer neural network was created. To perform training, we used a minibatch size of 500 and 5000 iterations, totalling at 166 epochs. By setting the weight-decay coefficient , the following training, validation and test accuracy vs. epoch curves for different values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







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 , 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**

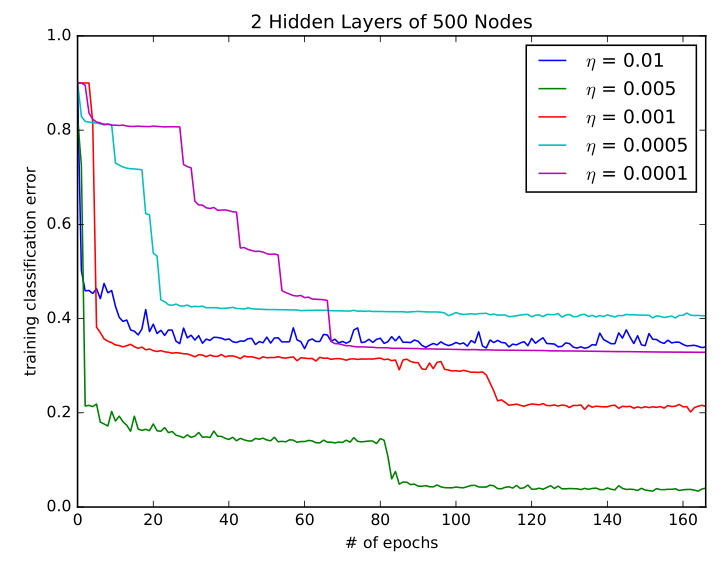
**1.2.1 – Number of hidden units**

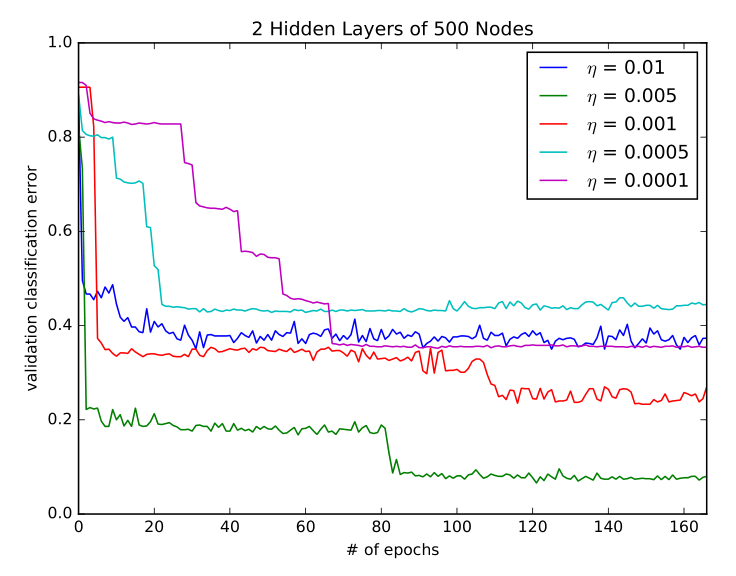
No weight decay was used in this model. The plots showing the training error and accuracies vs iterations (per epoch) can be seen in Appendix A. The following table summarizes the final values (best results bolded) for tuning the hyperparameters (where ):

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **# of nodes** | 100 | | | 500 | | | 1000 | | |
|  | 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.

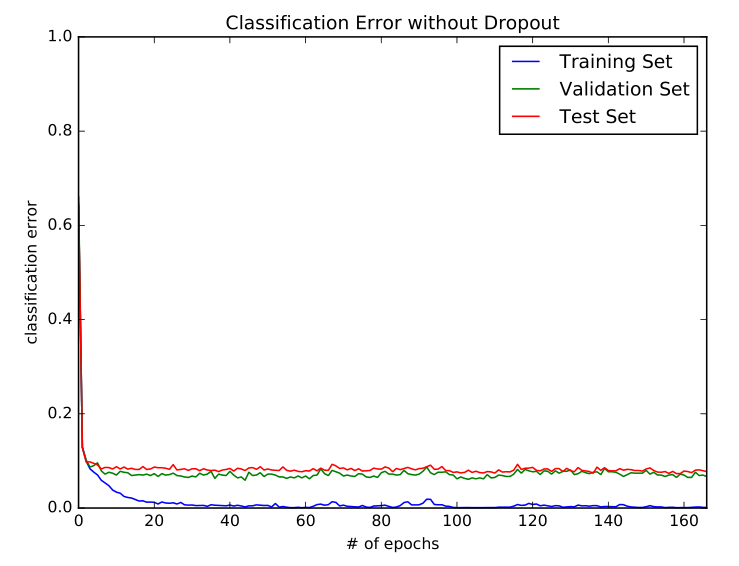
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0.01 | **0.005** | 0.001 | 0.0005 | 0.0001 |
| **Val. error** | 0.373 | **0.0800** | 0.270 | 0.445 | 0.354 |
| **Test error** | 0.358 | **0.0830** | 0.262 | 0.452 | 0.356 |

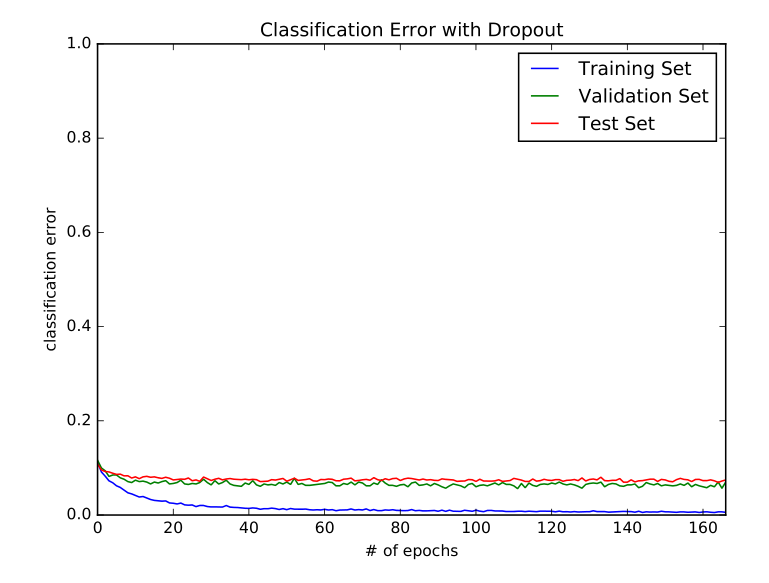
Comparing the results of the test set for in the 2-layer neural network, we see that the error is on par, if not a bit worse than the single layer neural network.

**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.

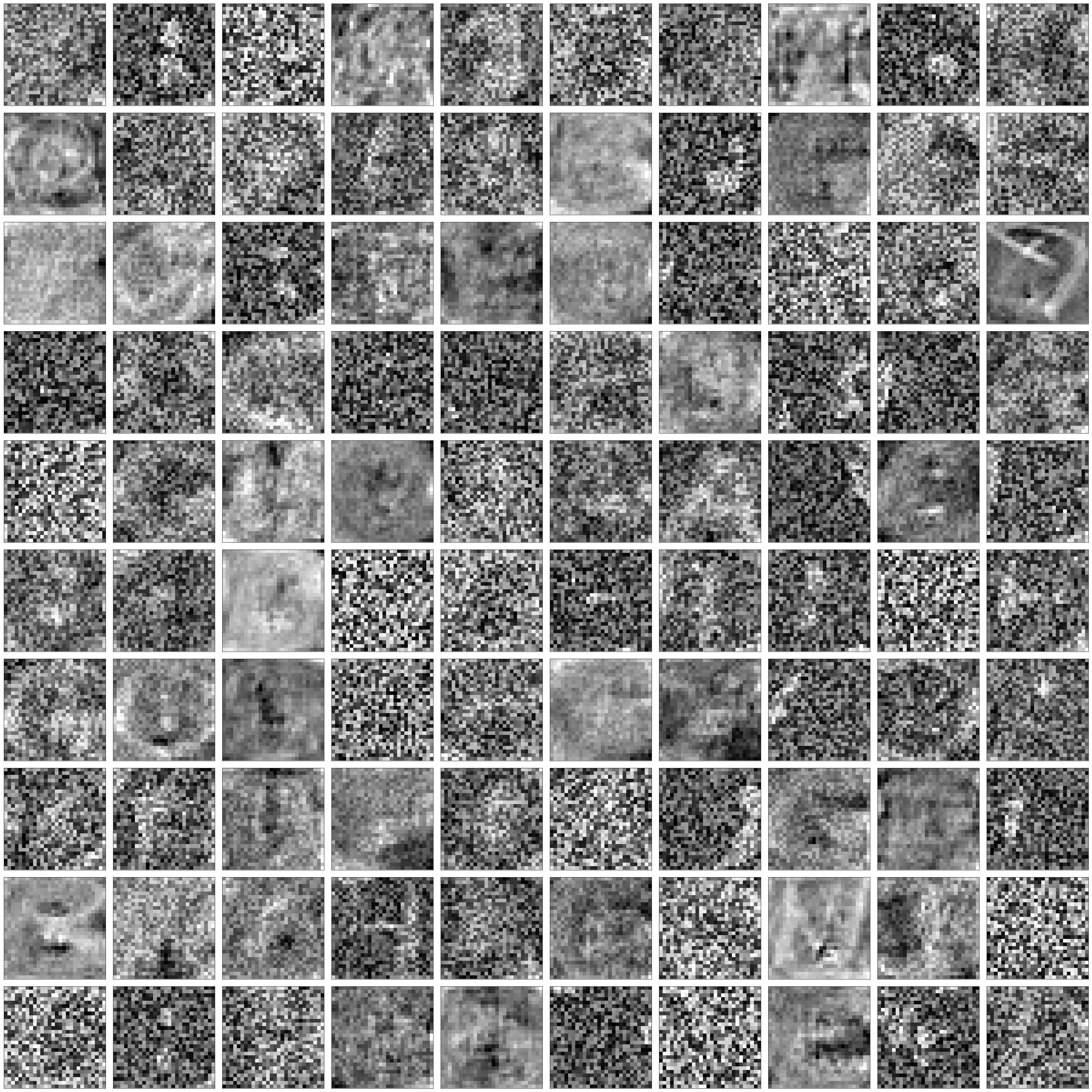




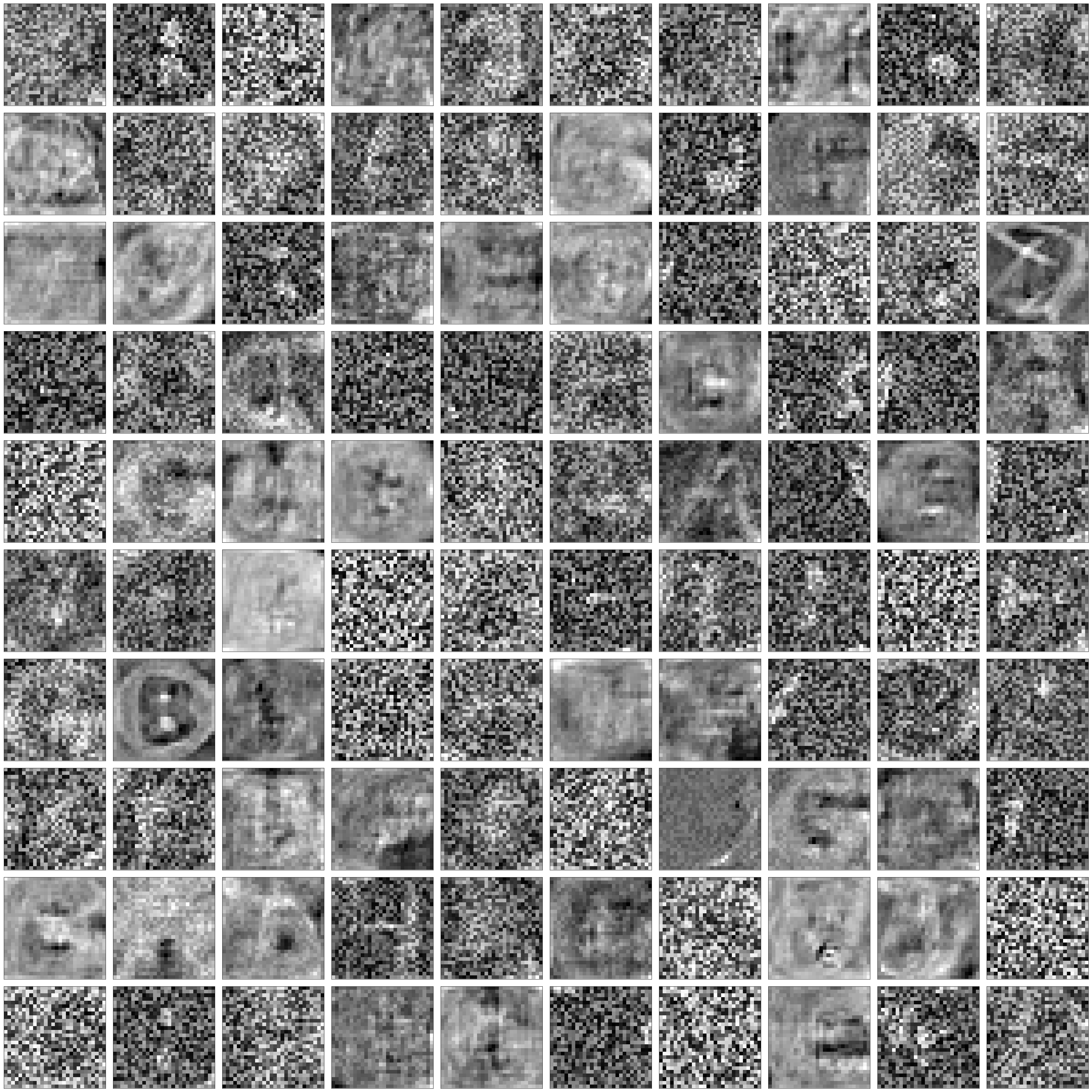
**1.3.2 – Visualization**

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.

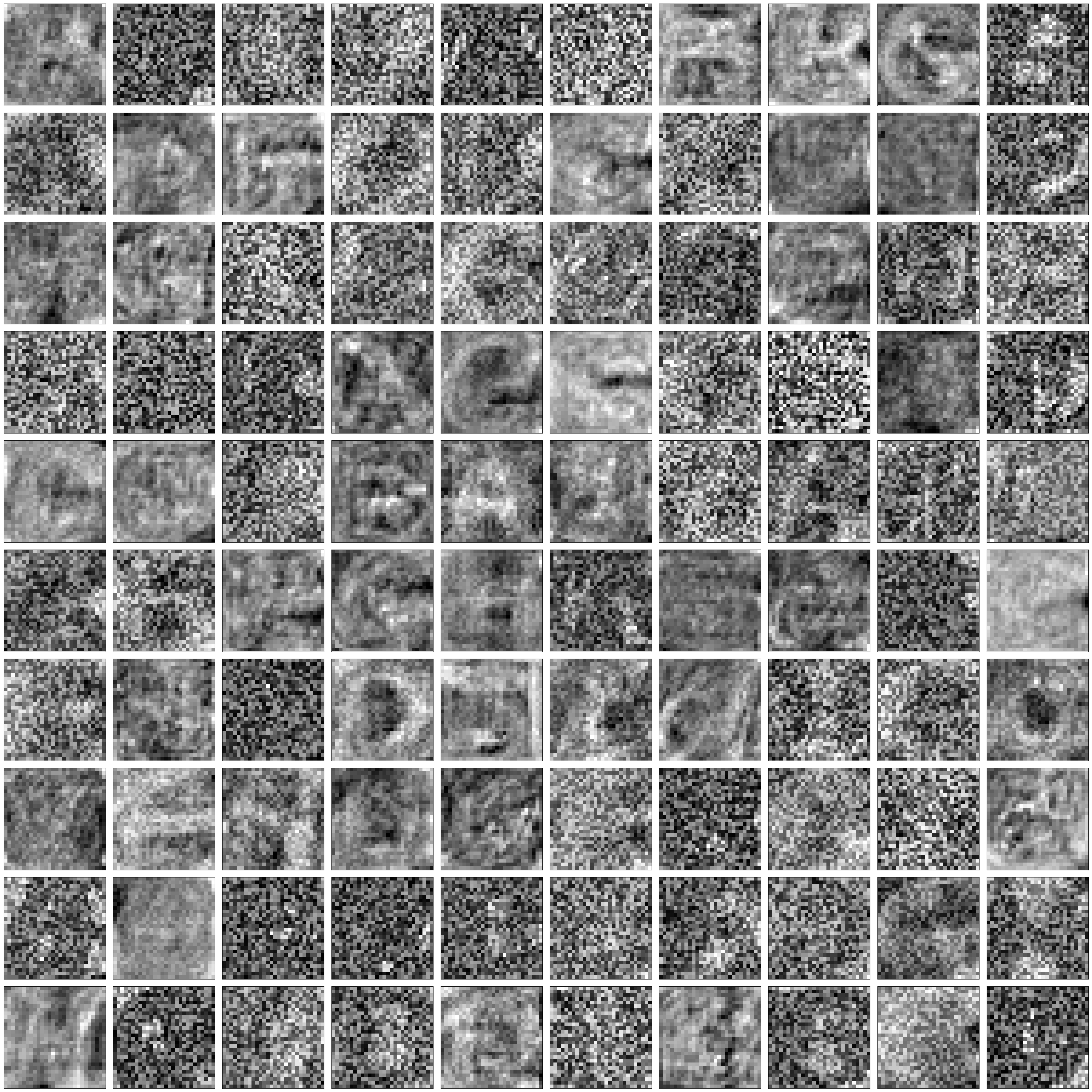
25% complete without dropout



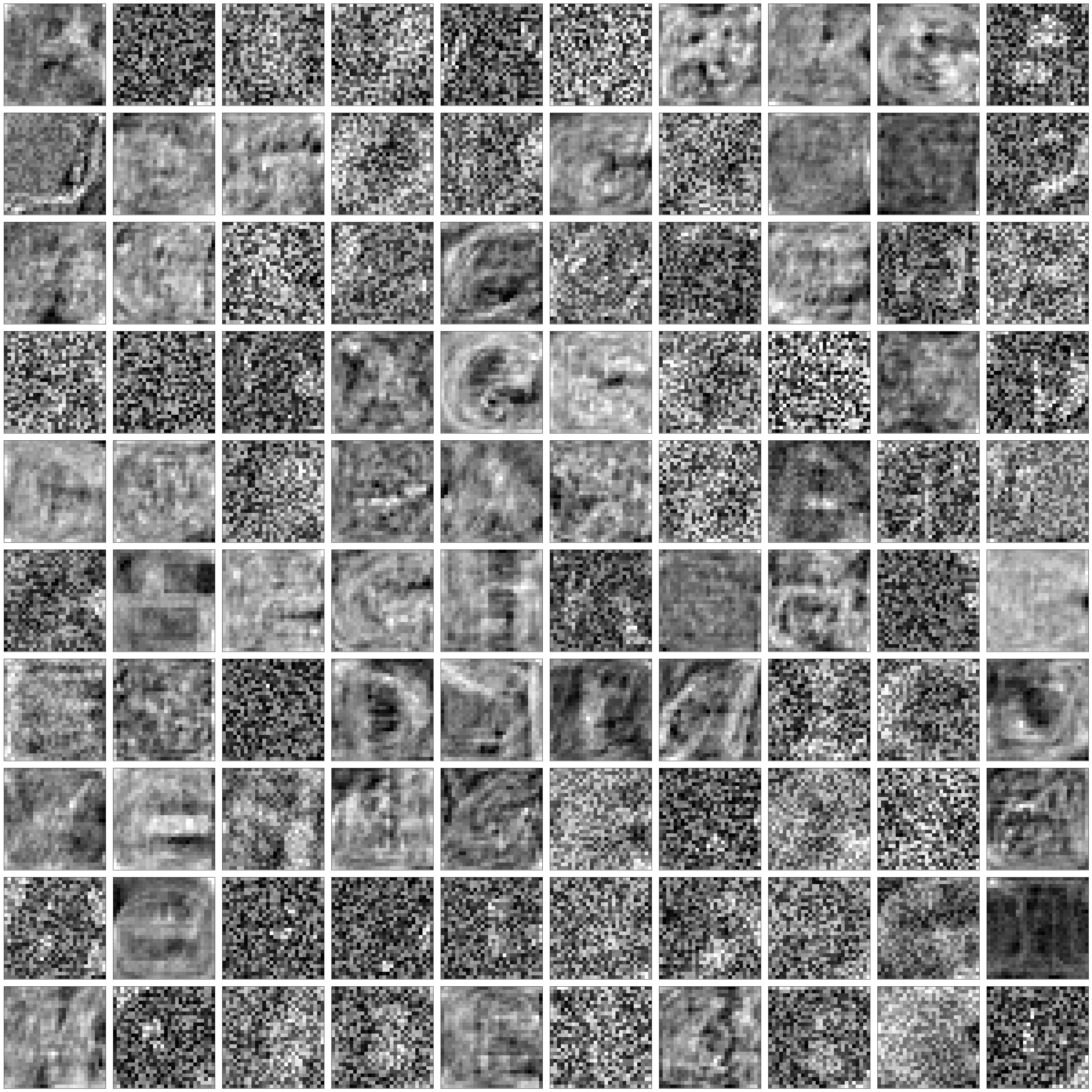
100% complete without dropout



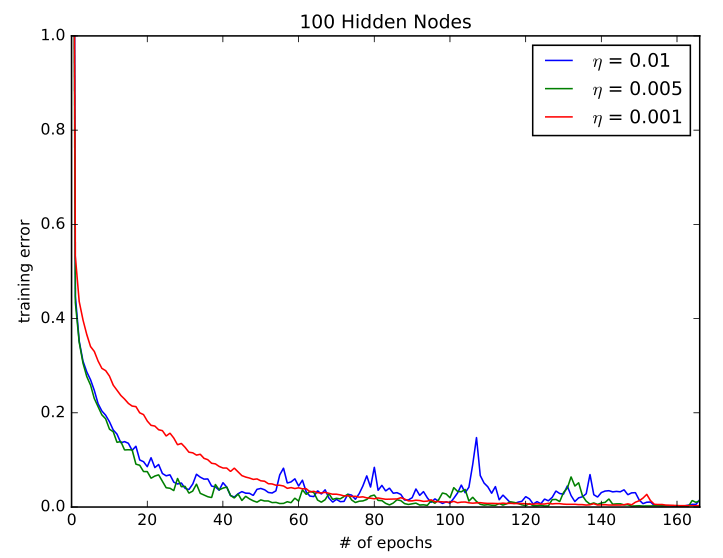
25% complete with dropout

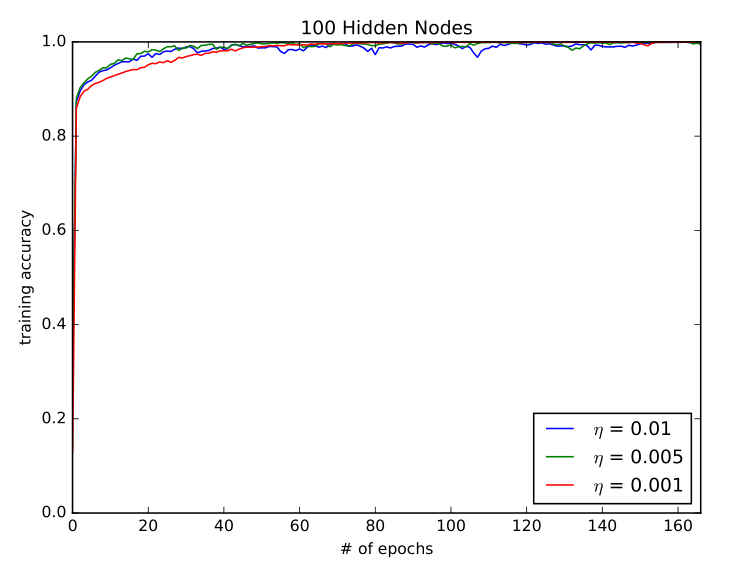


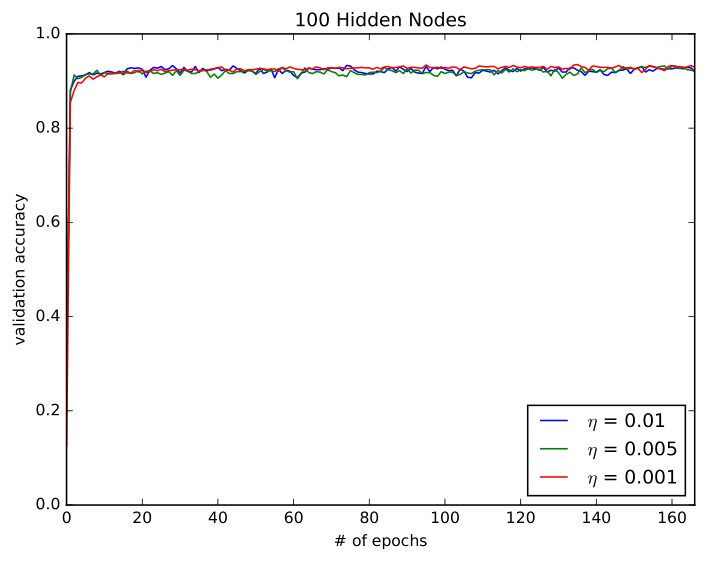
100% complete with dropout

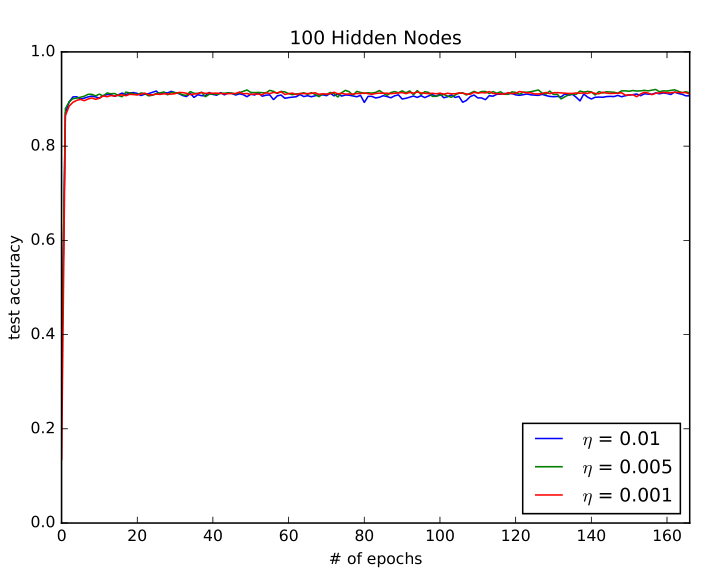


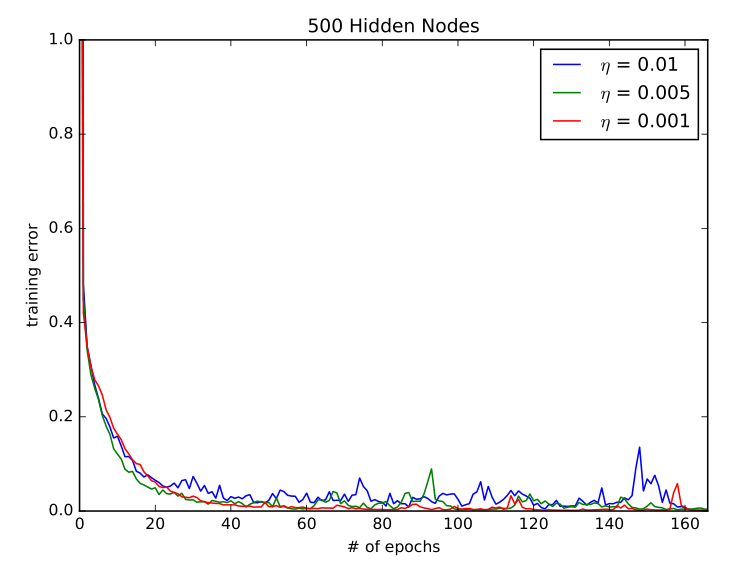
**GRAPHS for 1.2.1**

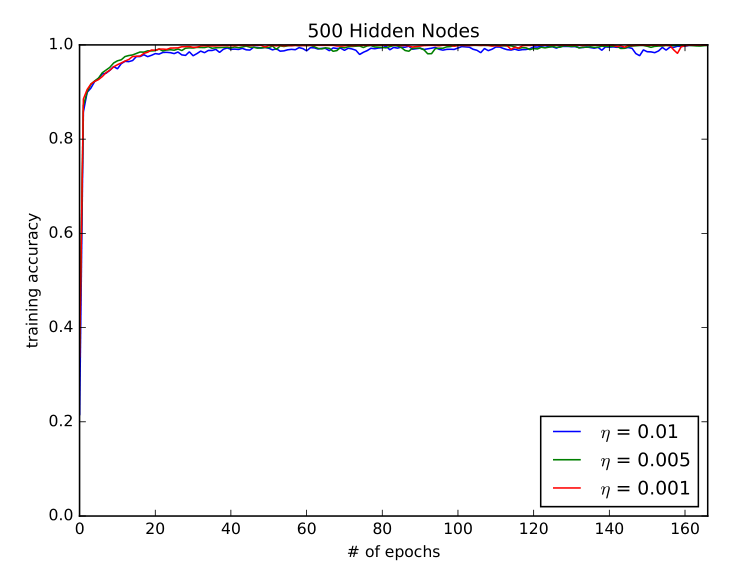


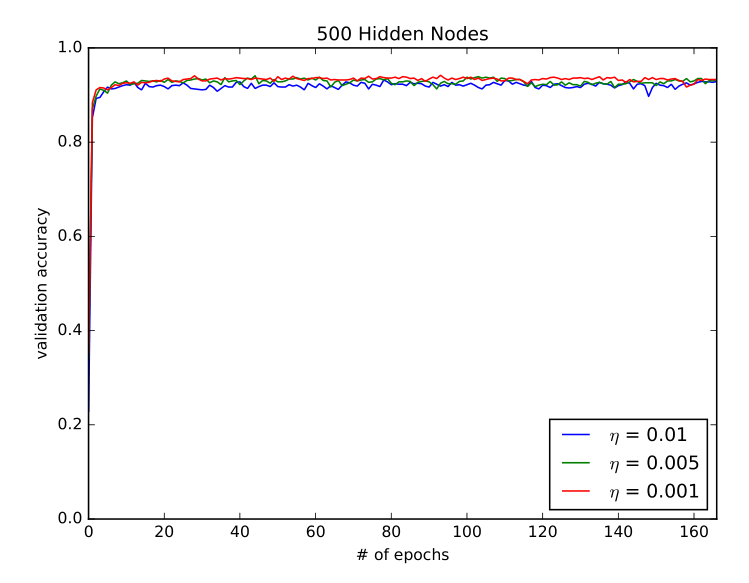


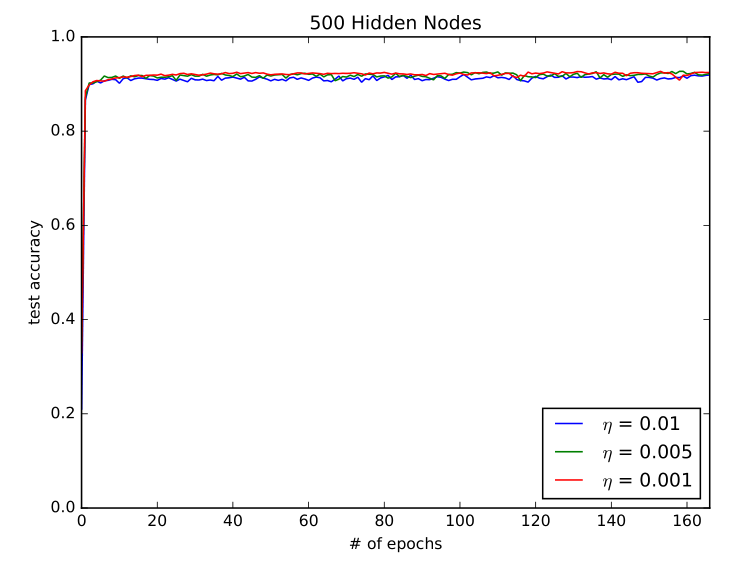


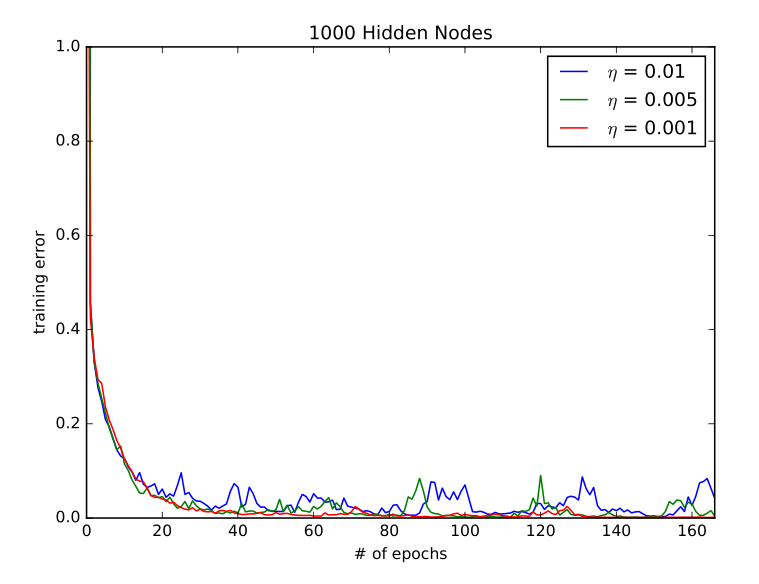


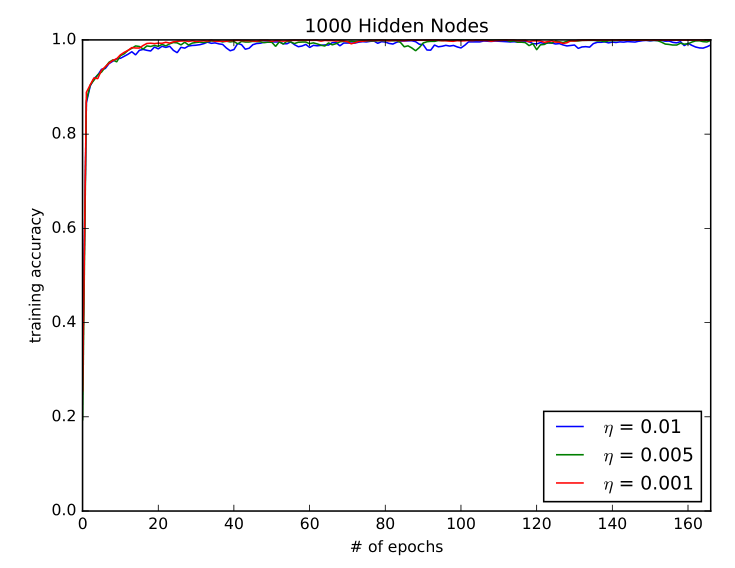


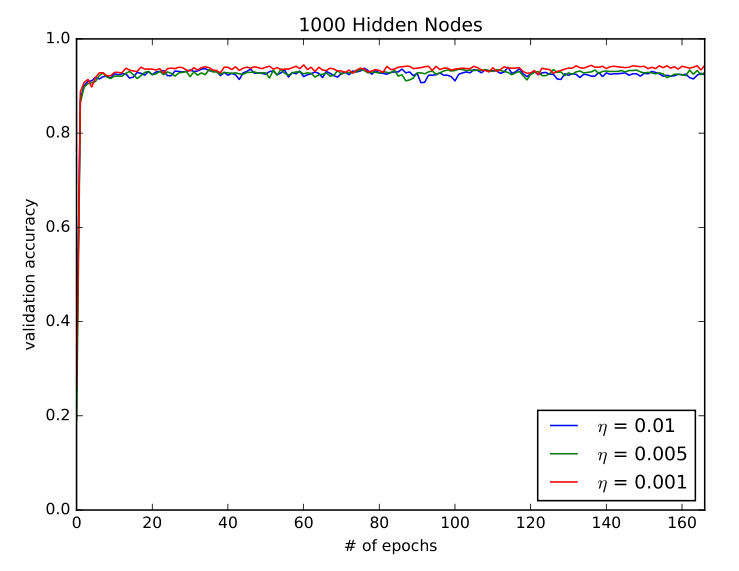


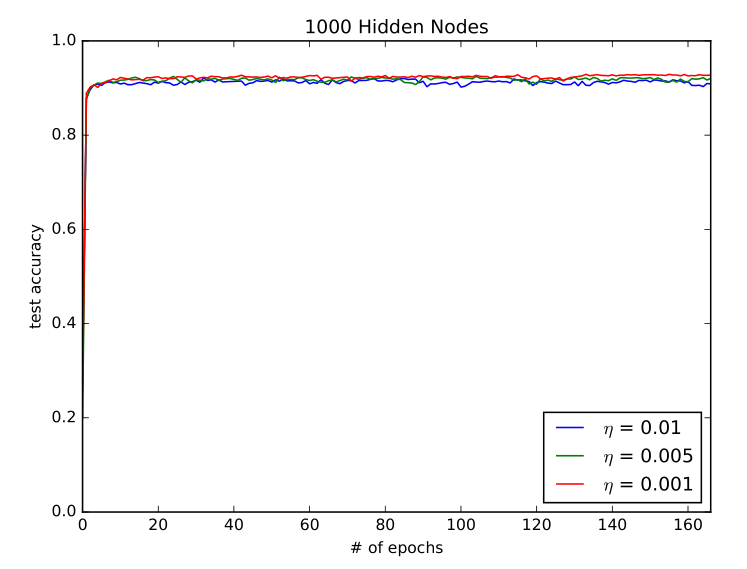












**Python Codes**

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() |