# ECE 521 Inference Algorithms and Machine Learning

Assignment #2

Student Name: Shengze Gao(ID: 1002935942)
Student Name: Jingxiong Luo(ID: 1002772332)

Feb 27th, 2017

Percentage of Contribution:
Jingxiong Luo: 50% Shengze Gao: 50%

## Table of Content

# 1. Logistic Regression

## 1.1 Binary cross-entropy loss

1.1.1

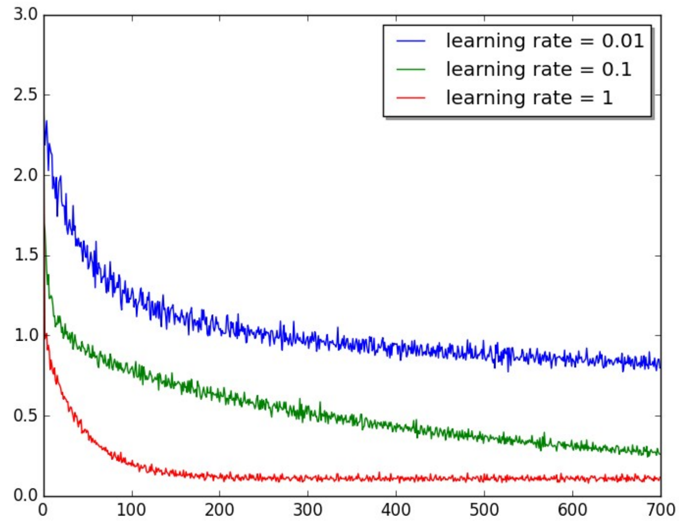
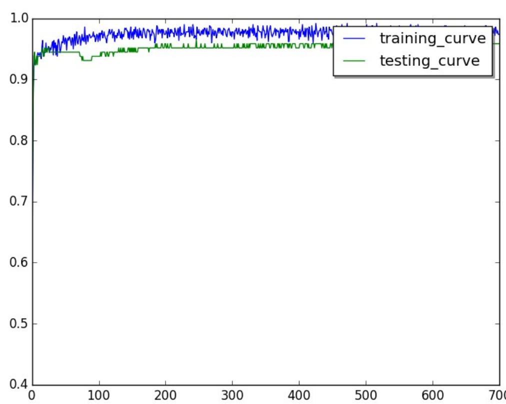



Figure 1.1.1.1

Figure 1.1.1.1 shows the best learning rate we find, learning rate = 1.



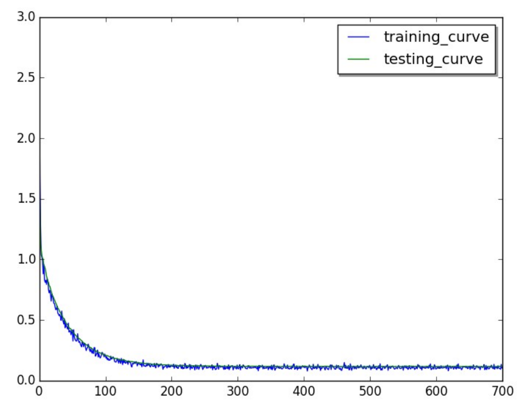




Figure 1.1.1.2 Figure 1.1.1.3

Figure 1.1.1.2 shows the classification accuracy vs the number of updates

Figure 1.1.1.3 shows the cross-entropy loss vs the number of updates

The best test classification accuracy obtained from the logistic regression model = 96.55%

1.1.2

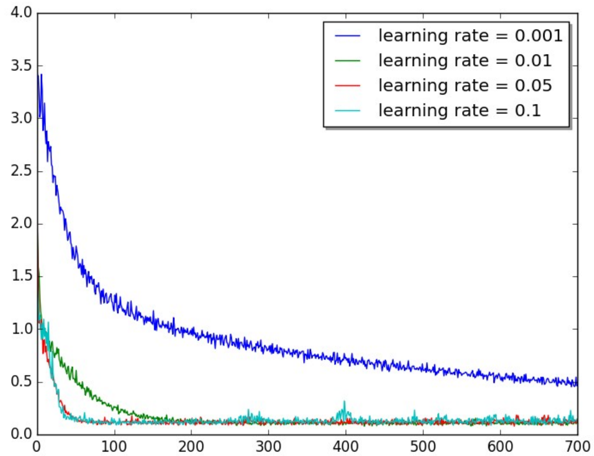



Figure 1.1.2.1

Figure 1.1.2.1 shows the best learning rate we find, learning rate = 0.05

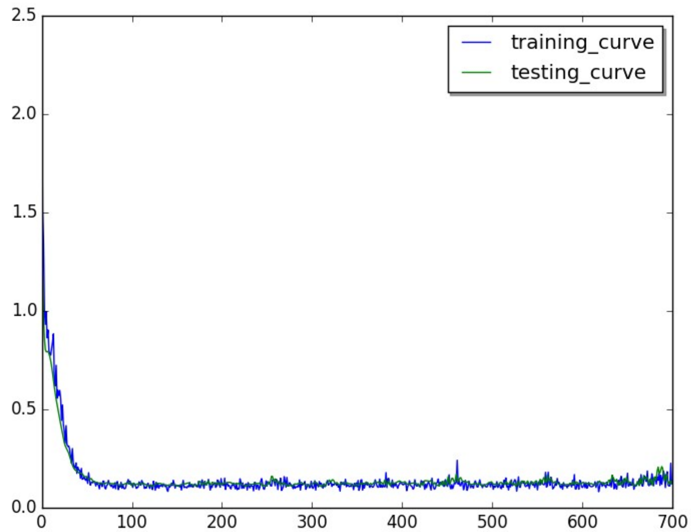



Figure 1.1.2.2

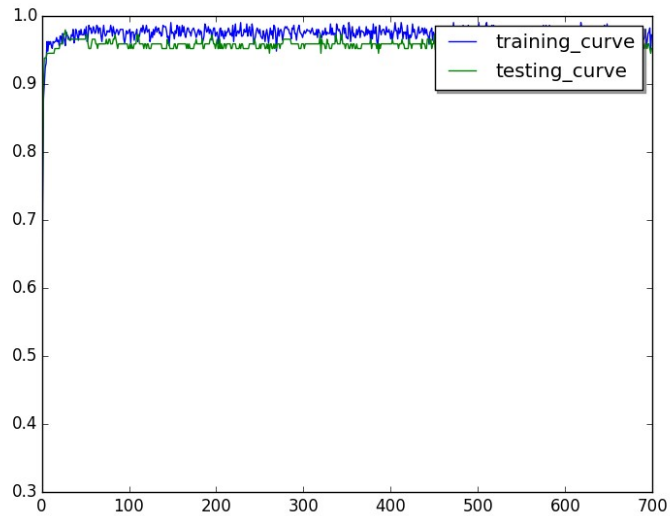



Figure 1.1.2.3

Figure 1.1.2.2 and figure 1.1.2.3 show the the best training and testing curves for both the cross-entropy loss and the classification accuracy. Compared with the SGD plots, we can see that the Adam-optimizer is faster than plain SGD, it takes fewer updates for Adam-optimizer to converge.

1.1.3

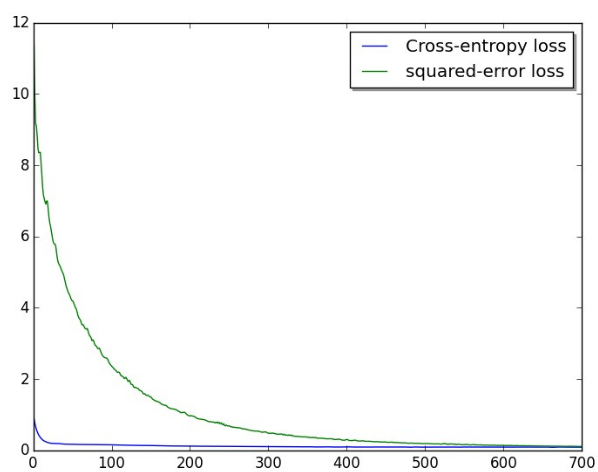



Figure 1.1.3.1

Figure 1.1.3.1 shows the test classification of the least squares solution vs the optimal logistic regression learnt

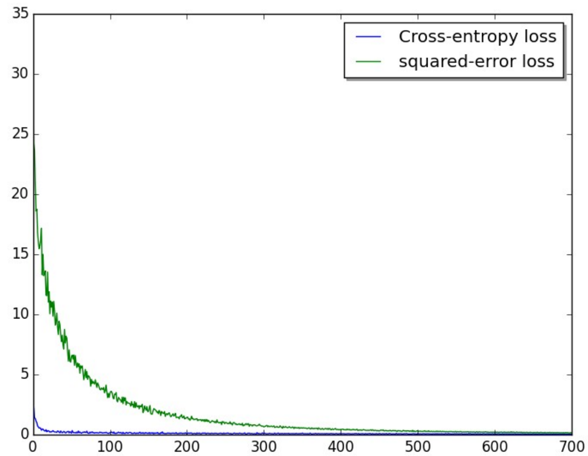



Figure 1.1.3.2

Figure 1.1.3.2 shows the training classification of the least squares solution vs the optimal logistic regression learnt

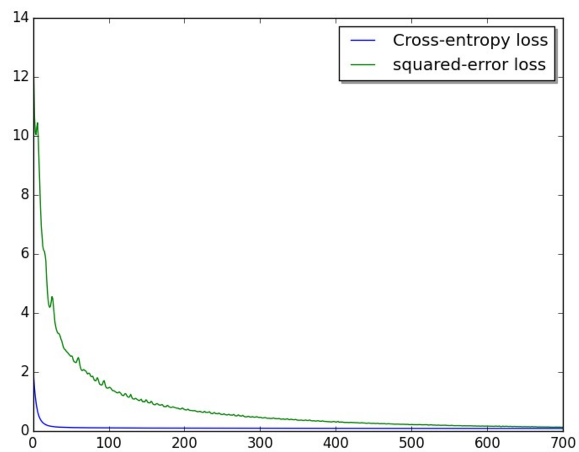



Figure 1.1.3.3

Figure 1.1.3.1 shows the validation classification of the least squares solution vs the optimal logistic regression learnt

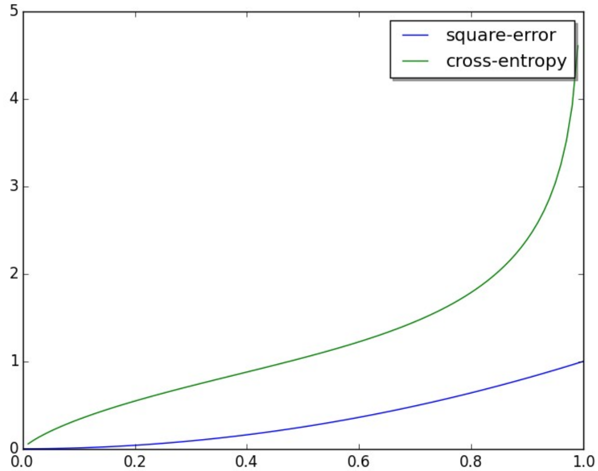



Figure 1.1.3.4

From figure 1.1.3.1, figure 1.1.3.2 and figure 1.1.3.3, we can see, during learning procedure, logistic regression is much faster than least squares solution. To explain the observation, we can see in the figure 1.1.3.4, we plot the cross-entropy loss vs squared-error loss as a function of the prediction ˆy within the interval [0, 1] and a dummy target y = 0. In this figure, when the prediction is far from target, which is 0, the cross-entropy loss is much steeper than the square_error loss. So in this area,the gradient of cross-entropy loss is larger, the learning process for cross-entropy loss is faster.

1.1.4

1.1.4

The Bernoulli distribution can be written as

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

The log-likelihood of the training data is

$$\ln [\prod_{m=1}^{M} p(y^{(m)}|x^{(m)}; W)]$$

$$= \sum_{m=1}^{M} y^{(m)} \ln [\hat{y}(x^n)] + \sum_{m=1}^{M} (1-y^{(m)}) \ln (1-\hat{y}(x^n))]$$

And the cross-entropy loss is

$$\sum_{m}^{M} \frac{1}{M} [-y^{(m)} \ln[\hat{y}(x^n)] - \sum_{m=1}^{M} (1-y^{(m)}) \ln (1-\hat{y}(x^n))]$$

So, minimizing the cross-entropy loss is equivalent to maximizing the log-likelihood of the training data

## 1.2 Multi-class classification

1.2.1

1.2.1

The expected loss:

$$E[L] = \sum_k \sum_j \int_{R_j} L_{kj} \cdot p(x, C_k)\, dx$$

Since the penalty $L_{jk}$ is equal for each class

To minimize the expected loss, if $p(x, C_k) > p(x, C_j)$ $\forall j \neq k$, then we should assign $x$ to class $C_k$

Using $p(x, C_k) = p(x) \cdot p(C_k|x)$, to minize the probability of making mistake, we assign each $x$ to the class for which the posterior probability $p(C_k|x)$ is the largest.

1.2.2

Loss Matrix $L = \begin{bmatrix} 0 & L_{21} & L_{31} & \cdots \\ L_{12} & 0 & & \\ L_{13} & & 0 & \\ \vdots & & & \ddots \end{bmatrix}$, $E[L] = \sum_k \sum_j \int_{R_j} L_{kj} P(x, C_k)\, dx$

We can write $P(x, C_k)$ as vector $\begin{bmatrix} P(x, C_1) \\ P(x, C_2) \\ \vdots \end{bmatrix}$

Then, we denote $E = L \cdot P(x, C_k)$

$$= \begin{bmatrix} 0 & L_{21} & L_{31} & \cdots \\ L_{12} & 0 & & \\ L_{13} & & 0 & \\ \vdots & & & \ddots \end{bmatrix} \cdot \begin{bmatrix} P(x, C_1) \\ P(x, C_2) \\ \vdots \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \end{bmatrix}$$

If $E_j > E_k$, $\forall k \neq j$, we should assign $x$ to class $C_j$ and the total loss is minimized.

1.2.3

We tune learning rate with three different values: 0.1, 0.01, 0.001 to observe which one gives the best cross-entropy loss.

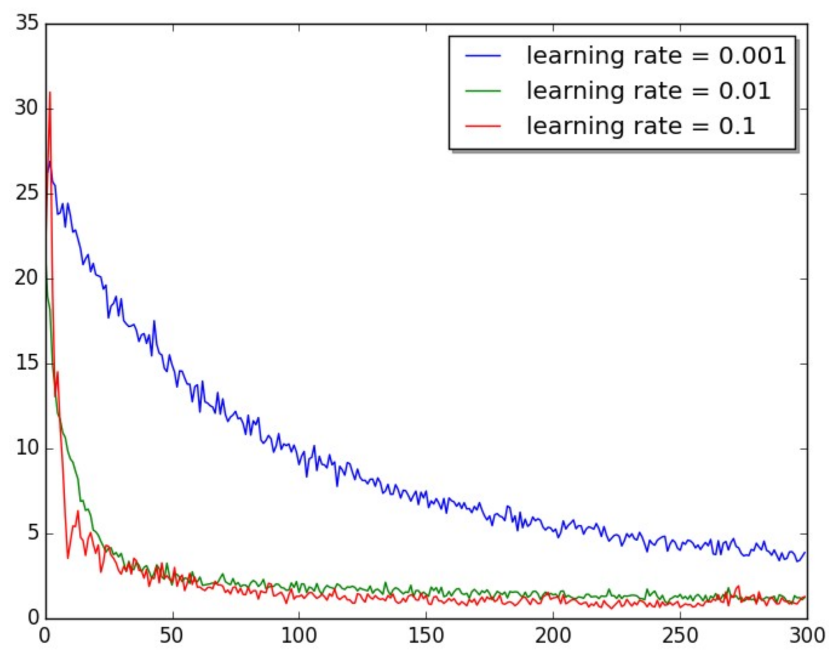



Figure 1.2.3.1

Since both 0.1 learning rate and 0.01 learning rate gives similar cross-entropy loss (better than

0.001 learning rate) and the cross-entropy loss under 0.01 learning rate are more stable than that under 0.1 learning rate, 0.01 learning rate should be considered as the best learning rate. The following plots show the training and testing curves for both cross-entropy loss and classification accuracy vs. the number of updates under 0.01 learning rate.

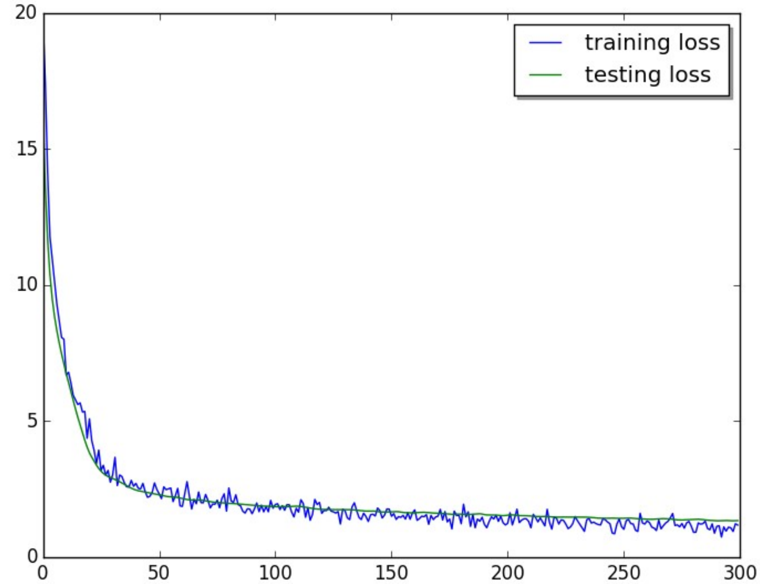



Figure 1.2.3.2: Cross-entropy loss vs. number of updates

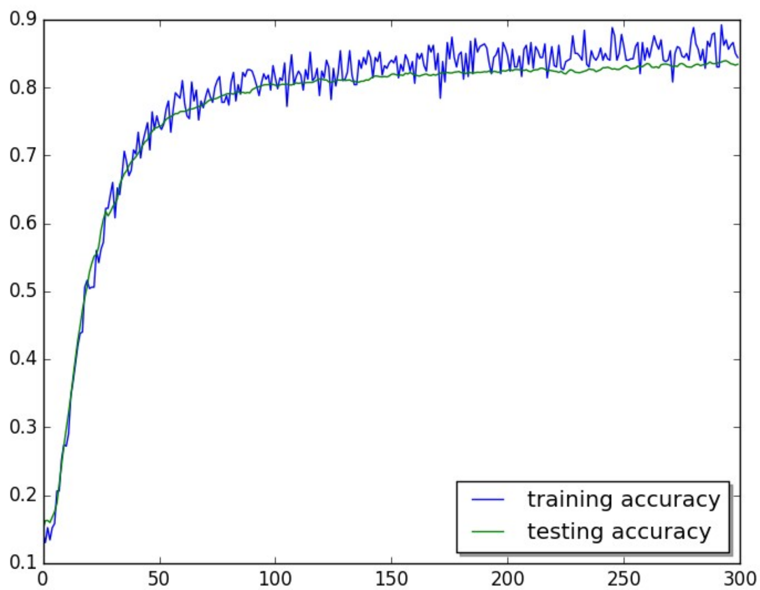



Figure 1.2.3.3: Classification accuracy vs. number of updates

The best test classification accuracy obtained from the logistic regression model is:

```
('best accuracy', 0.85866372980910421)
```

Recall from question 1.1.1, the best classification accuracy for binary-class problem is 96.55%. In this case, the best classification accuracy is 85.87% which is worse than the one of binary-class problem. This is probably due to the complexity of multi-class problem. In this case, the goal is to classify dataset into 10 different classes rather than 2 classes. Intuitively, the classification accuracy will decrease.

# 2. Neural Networks

## 2.1 Geometry of neural networks

2.1.1

Suppose there is a binary classification dataset with {0, 1} as target values. The optimal weights vector can be obtained when minimizing the loss function, L(W). We also have y_hat = $\sigma$ (WX + b). When L(W) approaches to 0, y_hat = $\sigma$ (WX + b) will be almost the same as y_target. Thus, predicted value y_hat will approach to target value 0 or 1:

If L(W) $\rightarrow$ 0, then y_hat = $\sigma$ (WX + b) $\rightarrow$ {0, 1}.

When y_hat $\rightarrow$ 0, WX + b $\rightarrow$ $-\infty$. Thus, W $\rightarrow$ $-\infty$.

When y_hat $\rightarrow$ 1, WX + b $\rightarrow$ $+\infty$. Thus, W $\rightarrow$ $+\infty$.

Thus, the L2 norm of the optimal weights $||W||_2 = \infty$

2.1.2

When there is a case that the binary classification dataset is linearly inseparable, it is impossible for the loss function L(W) to be minimized as close to zero. In another word, there always are many input data misclassified to the wrong class in a linearly inseparable binary classification. Thus, L(W) never be able to approach to zero when y_hat = $\sigma$ (WX + b) for a linearly inseparable classification.

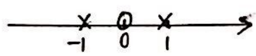
Similarly, suppose we have the target value as {0, 1}. The loss function will approach to a number larger than zero, while y_hat = $\sigma$ (WX+b) will not approach to the target value as {0, 1}. Thus, no matter y_target = 0 or y_target = 1, we always have $-\infty$ < WX+b < $+\infty$. So, the L2 norm of the optimal weights $||W||_2$ always be bounded, $||W||_2 < \infty$.

2.1.3

Consider a linearly inseparable binary classification dataset

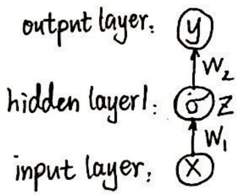
$x = [-1\ ,\ 0\ ,\ 1]$ which has a target value $y\_target = [1, 0, 1]$

It is standed by the following graph :




class 1 : 0
class 2 : X

Suppose there is a neural network shown as follows :




$$z = W_1 x + b_1 \qquad W_1 = \begin{bmatrix} W_a \\ W_b \\ W_c \end{bmatrix}$$

$$y = W_2 z + b_2 \qquad W_2 = [W_d\ W_e\ W_f]$$

$$Z = \sigma\left([-1\ 0\ 1]\begin{bmatrix} W_a \\ W_b \\ W_c \end{bmatrix} + [b_1]\right) = \sigma([-W_a + W_c + b_1]) = \sigma(-W_a + W_c + b_1)$$

$$\hat{y} = [\sigma(-W_a + W_c + b_1)]\,[W_d\ W_e\ W_f] + [b_2]$$

$$= [\sigma(-W_a+W_c+b_1)W_d + b_2 \quad \sigma(-W_a+W_c+b_1)W_e + b_2 \quad \sigma(-W_a+W_c+b_1)W_f + b_2]$$

When initialize : $W_a = 0$ , $W_c = +\infty$ ; $W_d = W_f = 1$ , $W_e = 0$ ; $b_1 = b_2 = 0$

$\hat{y} = y\_target = [1\ 0\ 1]$ , In this case $\|vec\{W^*\}\|_2 = \infty$.

We have an unbounded weight vector.

When initialize : $W_a = W_c = 0.5$ ; $W_d = W_f = 2$ , $W_e = 0$ ; $b_1 = b_2 = 0$

$\hat{y} = y\_target = [1\ 0\ 1]$ , in this case, $\|vec\{W^*\}\|_2 < \infty$

We have a bounded weight vector.

## 2.2 Feedforward fully connected neural networks

### 2.2.1

```python
def neural_network_model(d, nodes):

    hidden_layer_1 = {'weights': tf.Variable(tf.random_normal([784, nodes], stddev=3./(nodes+batch_size))),
                      'biases': tf.Variable(0.0, [nodes,])}
    output_layer = {'weights': tf.Variable(tf.random_normal([nodes, n_classes], stddev=3./(nodes+n_classes))),
                    'biases': tf.Variable(0.0, [n_classes,])}

    hidden_layer_1['weights'] = tf.cast(hidden_layer_1['weights'], tf.float64)
    hidden_layer_1['biases'] = tf.cast(hidden_layer_1['biases'], tf.float64)
    output_layer['weights'] = tf.cast(output_layer['weights'], tf.float64)
    output_layer['biases'] = tf.cast(output_layer['biases'], tf.float64)

    l1 = tf.add(tf.matmul(d, hidden_layer_1['weights']), hidden_layer_1['biases'])
    l1 = tf.nn.relu(l1)
    output = tf.matmul(l1, output_layer['weights']) + output_layer['biases']

    return output, hidden_layer_1['weights'], output_layer['weights']
def train_neural_network(x,n):
    train_err = []
    prediction, W1, W = neural_network_model(x, n_nodes_hl1)
    cost = tf.reduce_mean(tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(prediction,y) +lamda * (tf.reduce_mean(tf.square(W1)) + tf.reduce_mean(tf.square(W)))/2)
    optimizer = tf.train.AdamOptimizer(learning_rate=n).minimize(cost)

    number_epochs = 30

    init = tf.initialize_all_variables()
    sess.run(init)

    for epoch in range(number_epochs):

        idx = np.arange(0, len(trainData))
        np.random.shuffle(idx)
        for i in range(0, int(len(trainData) / batch_size)):
            batch1 = trainData[idx]
            batch2 = trainTarget[idx]
            batch1 = batch1[i * batch_size:(i + 1) * batch_size]
            batch2 = batch2[i * batch_size:(i + 1) * batch_size]
#            epoch_data = np.float64(epoch_data)
#            epoch_data = trainData.next_batch(batch_size)
#            epoch_target = trainTarget.next_batch(batch_size)
#            p, c, _ = sess.run([prediction, cost, optimizer], feed_dict = {x: batch1, y: batch2})

            , c, p = sess.run([optimizer, cost, prediction], feed dict={x: batch1, y: batch2})
```

Figure 2.2.1

Full version of the codes can be found in the appendix.

2.2.2

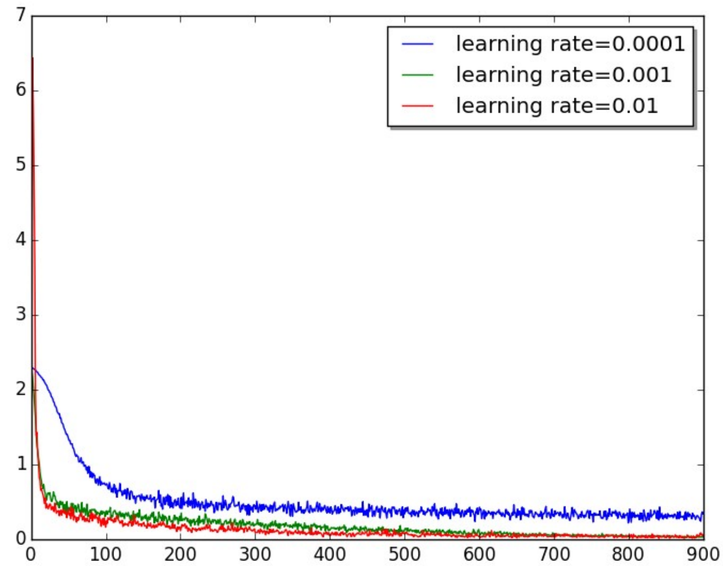



Figure 2.2.2.1

Figure 2.2.2.1 shows that we can the best learning rate we find = 0.001

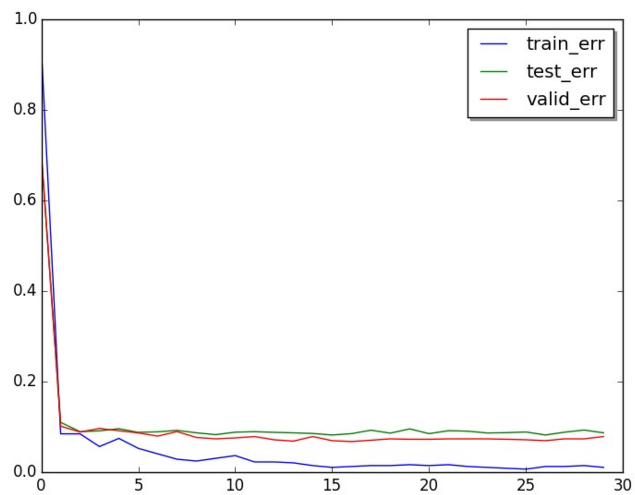



Figure 2.2.2.2 the classification error vs. the number of epochs

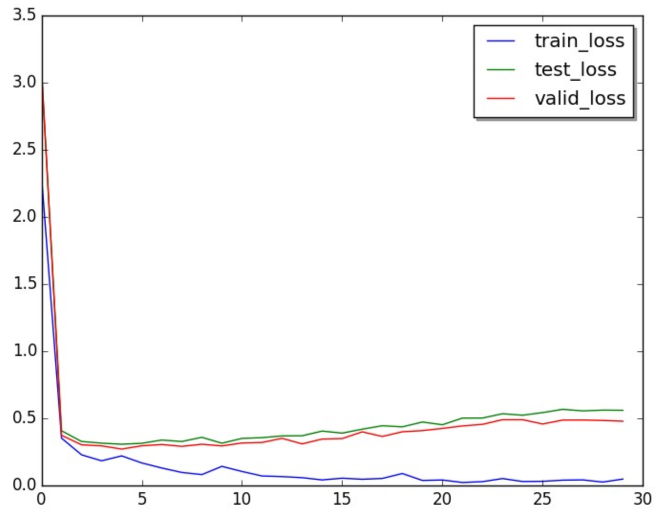



Figure 2.2.2.3 the cross-entropy loss vs. the number of epochs

We can see in the figure 2.2.2.2, the classification error decrease at first, and after about the 13th epoch, it doesn't change much. And in the figure 2.2.2.3, the cross-entropy loss decrease at first , and after the 4th epoch, it increase with every epoch.

2.2.3

According to figure 2.2.2.2, the early stopping point on the classification error plot is the 13th epoch.

At this epoch,

Training classification error = 0.04
Test classification error = 0.087
Validation classification error = 0.072

The early stopping points are not the same on the two plots, it's the 13th epoch in figure 2.2.2.2 and the 4th epoch in figure 2.2.2.3. Because the overfitting will not reflect immediately in the accuracy plot . For example, when a prediction changes from 0.9 to 0.8 (with target = 1), the cross-entropy loss will increase, but the classification accuracy will stay the same. The cross-entropy loss plot should be used for early stopping. Because in preventing overfitting, the cross-entropy loss plot is more accurate.

## 2.3 Effect of hyperparameters

### 2.3.1

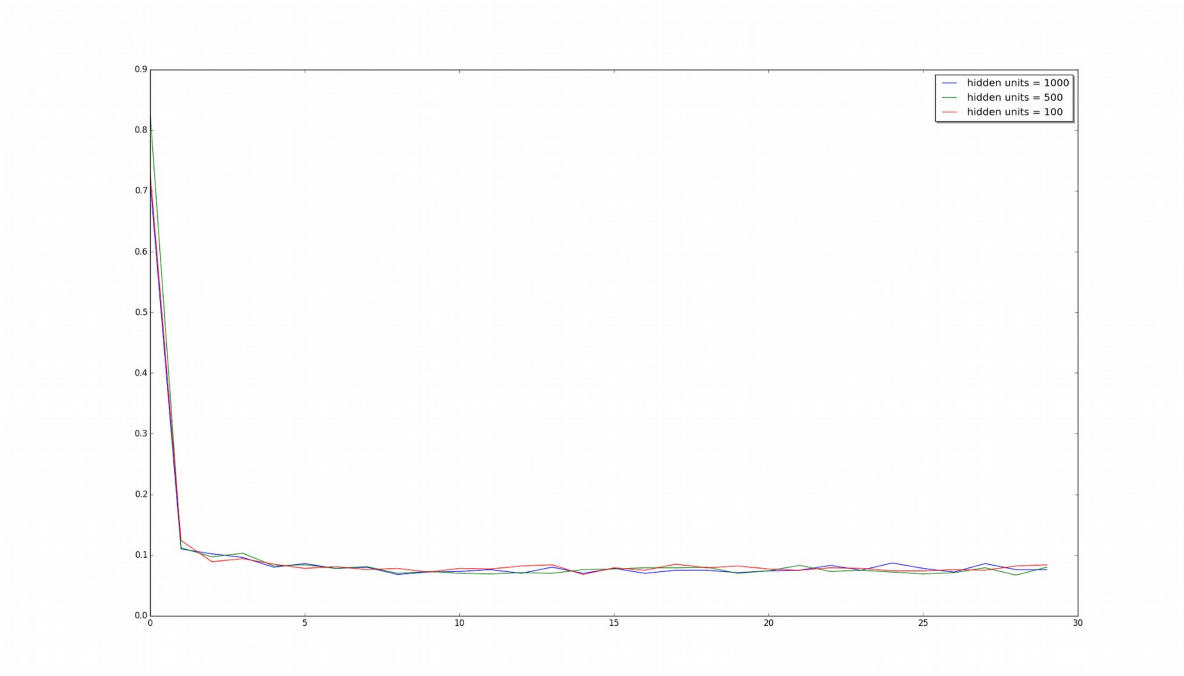



Figure 2.3.1 the classification error vs. the number of epochs

```
('1000 units, best validation error = ', 0.055000000000000049)
('500 units, best validation error = ', 0.057000000000000051)
('100 units, best validation error = ', 0.065999999999999948)
```

When works with 1000 units we have the best validation error. Then use it for classifying the test set, the test classification error = 0.082

2.3.2

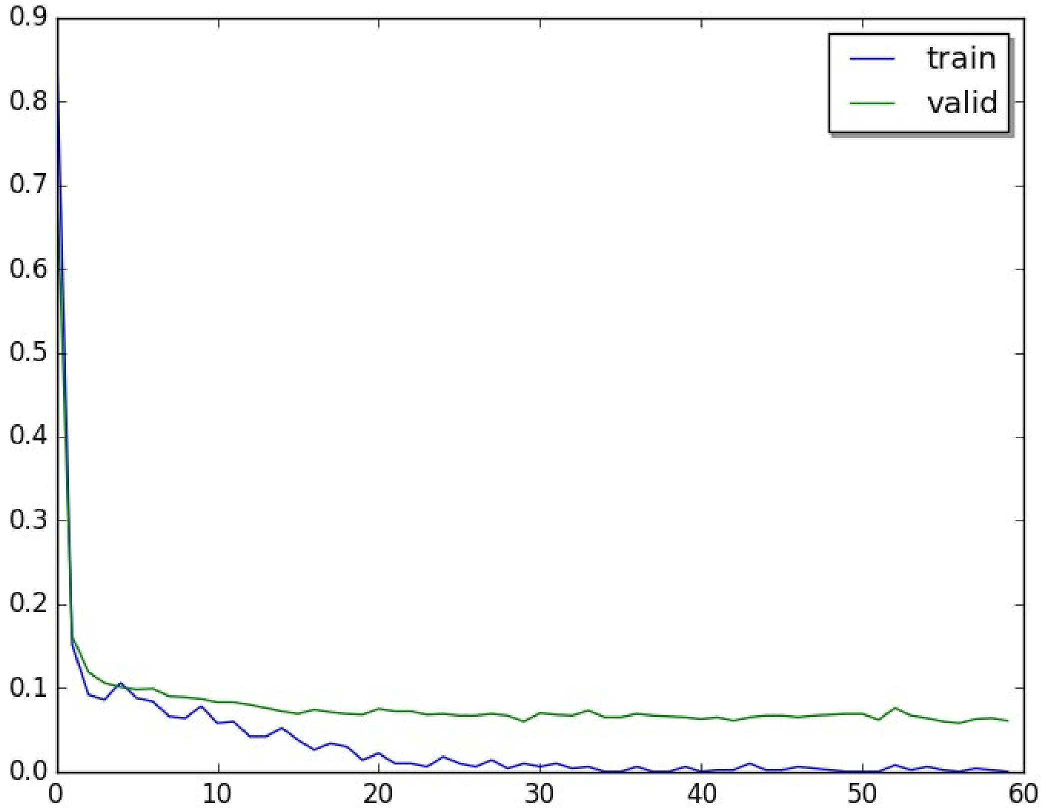



Figure 2.3.2 training and validation classification errors vs. the number of epochs(2 layers)

The final validation error = 0.068
Using the test set, we get test accuracy = 0.08
Compared with the architecture with the one-layer case, the two layers architecture has almost the same (a little better) test accuracy. But the  two layer architecture is more complex than the one layer architecture, it more likely to be overfitting.

## 2.4 Regularization and visualization

### 2.4.1

The number of training and validation classification errors vs. the number of epochs with dropout is shown as follows:

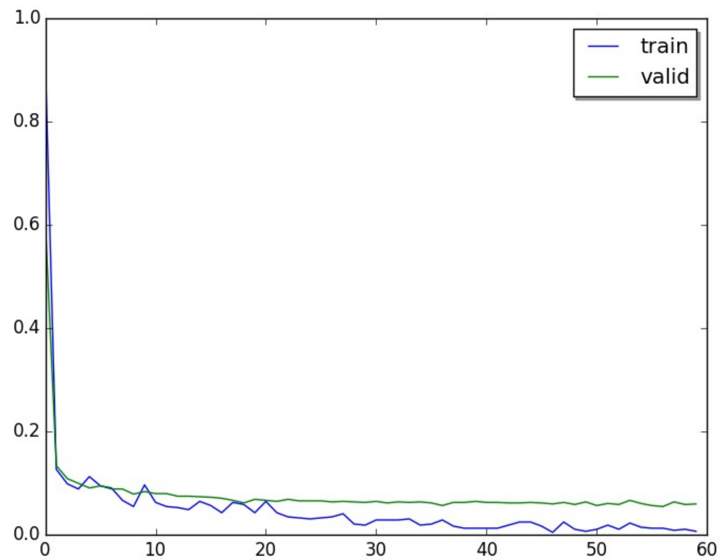



Figure 2.4.1.1: Classification errors vs. the number of epochs with dropout

The training error and validation error in this case is shown as follows:

```
('training errors = ', 0.0060000000000000053)
('validation errors =', 0.059000000000000052)
```

In order to compare it to the one without dropout, the following plot shows the number of training and validation classification errors vs. the number of epochs without dropout:

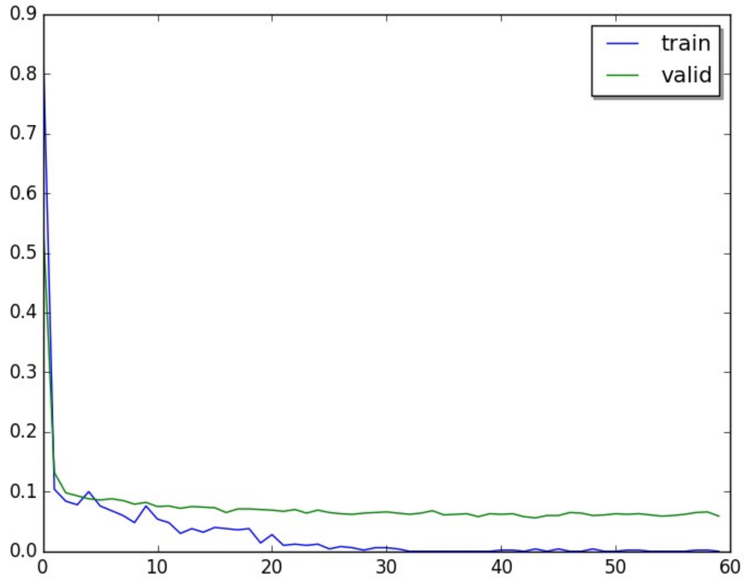



Figure 2.4.1.2: Classification errors vs. the number of epochs without dropout

The training error and validation error in this case is shown as follows:

```
('training errors = ', 0.0)
('validation errors =', 0.06299999999999945)
```

Comparing the case with and without dropout, it is obviously that the training error with dropout will never decrease down to 0, whereas the training error without dropout decreases to 0 at about 30 epoch. Meanwhile, the validation error with dropout is smaller than the validation error without dropout. Thus, dropout can effectively reduce overfitting.

2.4.2

We set five checkout points (0%, 25%, 50%, 75%, 100% of complement of training) for both cases of with and without dropout.

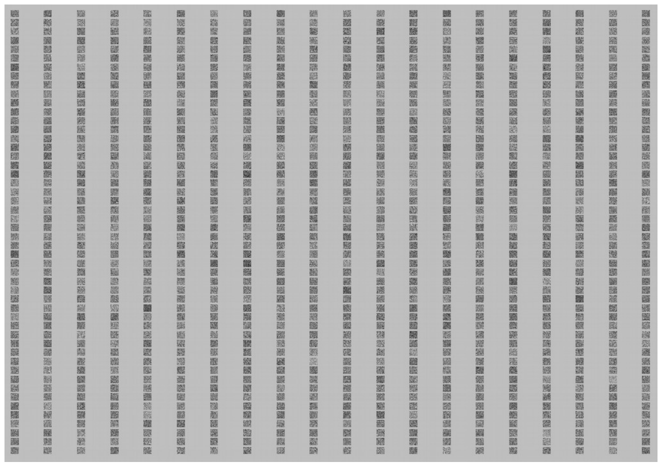


Figure 2.4.2.1 Visualization of the case with dropout (0% of complement)

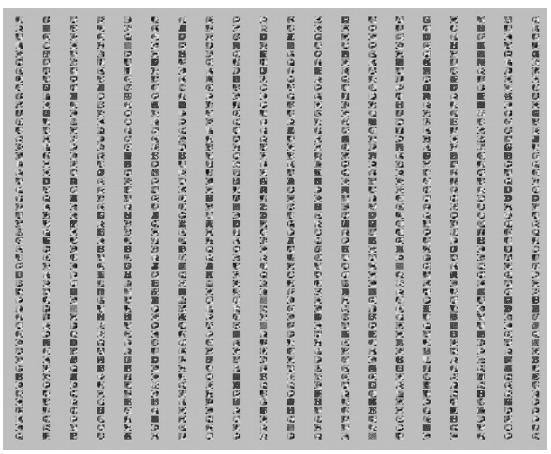


Figure 2.4.2.2 With dropout (25%)

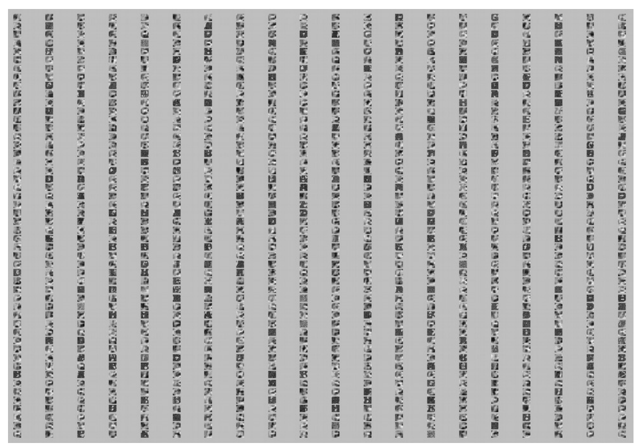


Figure 2.4.2.3 With dropout (50%)

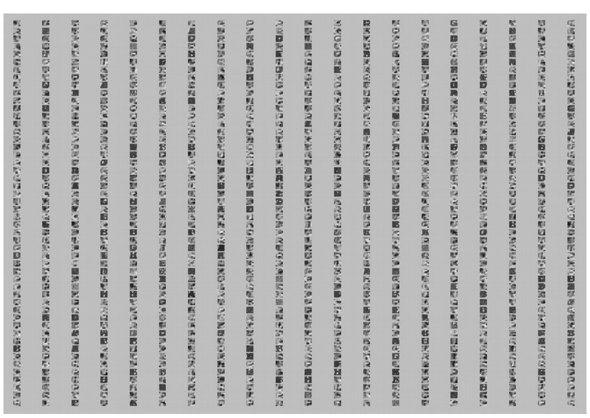


Figure 2.4.2.4 With dropout (75%)

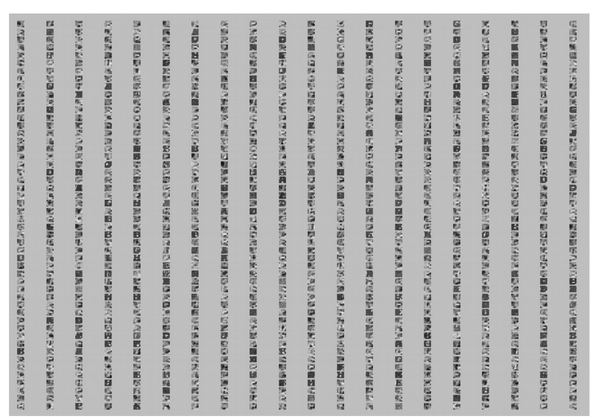


Figure 2.4.2.5 With dropout (100%)

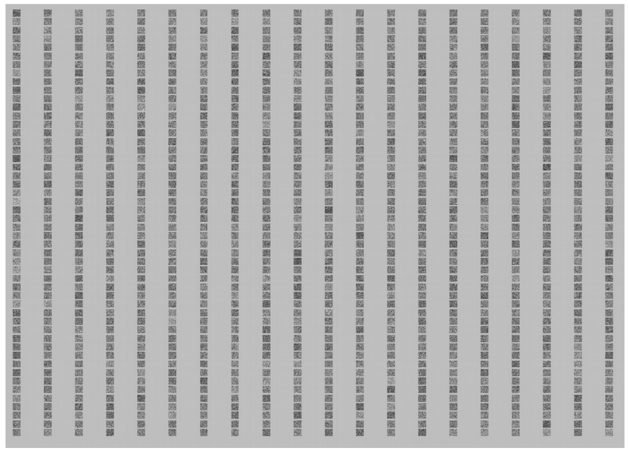


Figure 2.4.2.6 Visualization of the case without dropout (0% of complement)

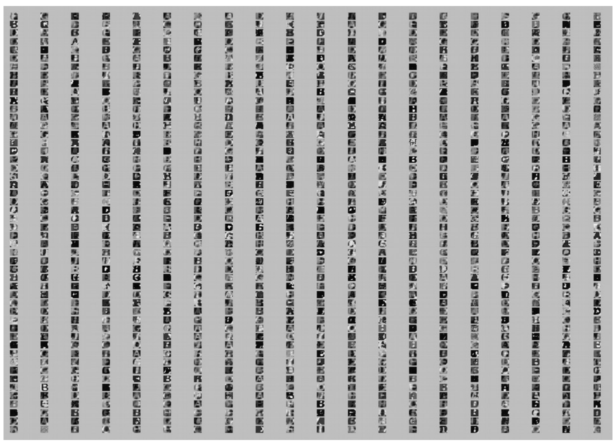


Figure 2.4.2.7 Without dropout (25%)

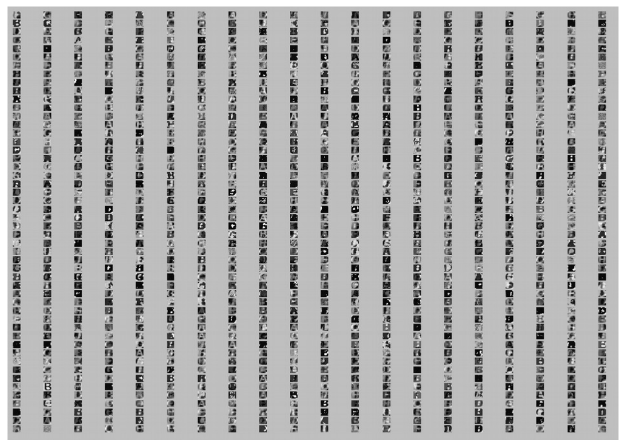


Figure 2.4.2.8 Without dropout (50%)

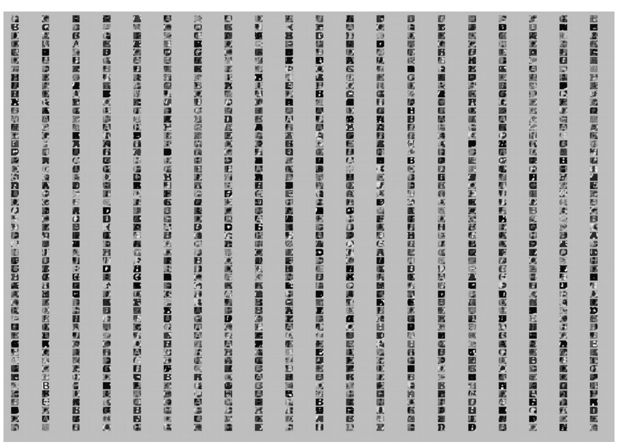


Figure 2.4.2.9 Without dropout (75%)

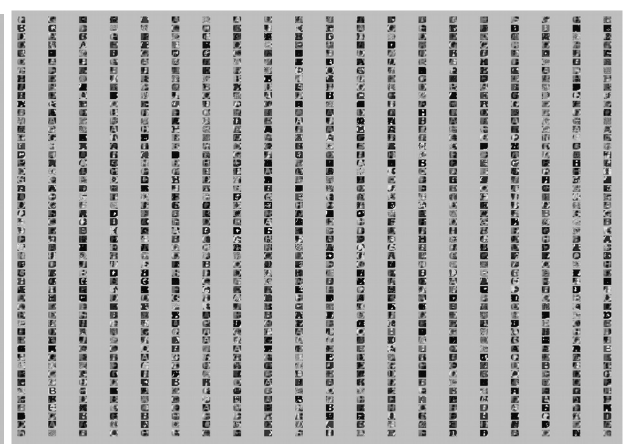


Figure 2.4.2.10 Without dropout (100%)

Deriving from the plots shown before, as the increasing of the percentage of complement, the visualization figures becomes clearer and clearer in both of the two cases (with and without dropout). Comparing these two models, the visualization of the case without dropout is distinctively clearer than the one with dropout. This is probably due to that the dropout may reduce the chance of neurons to be trained.

## 2.5 Exhaustive search for the best set of hyperparameters

### 2.5.1

| Model | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Test Error | 8.25% | 11.3% | 8.0% | 8.22% | 9.47% |
| Validation Error | 7.79% | 10.6% | 7.29% | 7.19% | 8.39% |
| Layers | 2 | 3 | 3 | 3 | 4 |
| Architecture | [209,409] | [211,305,234] | [147,181,306] | [360,468,165] | [155,293,161,353] |
| Weight Decay | 0.00214 | 0.00151 | 0.00096 | 0.00197 | 0.00035 |
| Learning Rate | 0.00192 | 0.00588 | 0.00413 | 0.00225 | 0.0069 |
| Dropout | NO | YES | NO | NO | NO |
| Keep Probability | - | 0.186 | - | - | - |

Numpy random seed = 1002772332
Tensorflow random seed = 1002935942

### 2.5.2

Best architecture I find:

- 5 hidden layers (496, 276, 486, 299, 376)
- No dropout
- Weight Decay Coefficient = 0.00091
- Learning Rate = 0.00292
- Validation Accuracy = 93.0%, Test Accuracy = 93.6%

# Appendix

1. Logistic regression

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

with np.load("notMNIST.npz") as data:
    Data, Target = data ["images"], data ["labels"]
    posClass = 2
    negClass = 9
    dataIndx = (Target==posClass) + (Target==negClass)
    Data = Data[dataIndx].reshape(-1, 784) / 255
    Target = Target[dataIndx].reshape(-1, 1)
    Target[Target==posClass] = 1
    Target[Target==negClass] = 0
    np.random.seed(521)
    randIndx = np.arange(len(Data))
    np.random.shuffle(randIndx)
    Data, Target = Data[randIndx], Target[randIndx]
    trainData, trainTarget = Data[:3500], Target[:3500]
    validData, validTarget = Data[3500:3600], Target[3500:3600]
    testData, testTarget = Data[3600:], Target[3600:]

sess = tf.InteractiveSession()

lamda = 0.01

batch_size = 500

number_updates = []
train_err0 = []
train_err1 = []
train_err2 = []
train_accuracy = []
test_err = []
test_accuracy = []

def buildGraph(N,lamda):
    # Variable creation
    W = tf.Variable(tf.truncated_normal(shape=[784,1],stddev=0.5), name='weights')
    b = tf.Variable(0.0, name='biases')
    X = tf.placeholder(tf.float64, [None,784], name='input_x')
    y_target = tf.placeholder(tf.float64, [None,1], name='target_y')

    W = tf.cast(W, tf.float64)
    b = tf.cast(b, tf.float64)
    # Graph definition
    y_logits = tf.matmul(X, W) + b
    y_predicted = tf.sigmoid(y_logits)
    # Error definition
    sigmoidCrossEntropyError = tf.reduce_mean(tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(y_logits,
y_target), name='sigCroEntroError') + lamda * tf.reduce_sum(tf.square(W))/float(2), name='mean_error')
#    meanSquaredError = 1/2*tf.reduce_mean(tf.reduce_mean(tf.square(y_predicted - y_target),
reduction_indices=1, name='squared_error') + lamda * tf.reduce_mean(tf.matmul(tf.transpose(W),W)),
name='mean_squared_error')
```

```
    # Training mechanism
    optimizer = tf.train.GradientDescentOptimizer(learning_rate = N)
    train = optimizer.minimize(loss=sigmoidCrossEntropyError)
    return W, b, X, y_target, y_logits, y_predicted, sigmoidCrossEntropyError, train


def ClassAccuracy(Data, Target, currentW, currentb):
    S = tf.shape(Target)[0].eval()
    Target = tf.reshape(Target,[S]).eval()
    Data = tf.cast(Data,tf.float64)
    y_predicted_v = tf.sigmoid(tf.matmul(Data,currentW) + currentb)
    y_predicted_v = tf.reshape(y_predicted_v, [S]).eval()
    y = np.zeros(S)
    for i in range(S):
        if (y_predicted_v[i] >= 0.5):
            y[i] = 1
        else:
            y[i] = 0
        y[i] = abs(Target[i]-y[i])
    z = 1 - np.sum(y)/np.shape(Target)[0]
    return z


def runMult(N):
    train_err1 = []
    #Build computation graph
    W, b, X, y_target, y_logits, y_predicted, sigmoidCrossEntropyError, train = buildGraph(N,lamda)
    #Initialize session
    init = tf.initialize_all_variables()
    sess.run(init)
    # Training model
    for step in range(0,100):

        idx = np.arange(0,3500)
        np.random.shuffle(idx)
        for index in range(0,int(3500/batch_size)):

            batch1 = trainData[idx]
            batch2 = trainTarget[idx]
            batch1 = batch1[index*batch_size:(index+1)*batch_size]
            batch2 = batch2[index*batch_size:(index+1)*batch_size]
            _, err, currentW, currentb, yhat_x, yhat = sess.run([train, sigmoidCrossEntropyError, W, b, y_logits,
y_predicted], feed_dict={X: batch1, y_target: batch2})

            if (1):
#             if not ((int(3500/batch_size)*step+index) % 5) or int(3500/batch_size)*step+index < 10 :
#                 plt.plot(yhat, 'o')
                print("Iter: %3d, ERR-train: %4.2f"%(int(3500/batch_size)*step+index, err))
                number_updates.append (int(3500/batch_size)*step+index)
                train_err1.append (err)

                train_accuracy.append (ClassAccuracy(batch1, batch2, currentW, currentb))
#                 print(train_accuracy)
#    plt.plot(number_updates, train_err1, '-')
#    plt.plot(number_updates, train_accuracy, '-')
#    plt.show()
```

```
# Testing model

                errTest = sess.run(sigmoidCrossEntropyError, feed_dict= {X: testData, y_target: testTarget})
                print("Iter: %3d, ERR-test: %4.2f"%(int(3500/batch_size)*step+index, errTest))
                test_err.append (errTest)
                test_accuracy.append (ClassAccuracy(testData, testTarget, currentW, currentb))
#                 print(test_accuracy)

#    plt.plot(number_updates, test_err, '-')

#    plt.plot(number_updates, test_accuracy, '-')
    return train_err1,test_err,train_accuracy, test_accuracy


# Tuning learing rate
#train_err0,_ = runMult(0.01)
#train_err1,_ = runMult(0.1)
train_err2,test_err,train_accuracy, test_accuracy = runMult(1)
#train_err.append(runMult(0.1))
#plt.plot(train_err0, label='learning rate = 0.01')
#plt.plot(train_err1, label='learning rate = 0.1')
#plt.plot(train_err2, label='learning rate = 1')
#legend = plt.legend(loc='upper right', shadow=True)
#plt.show()

# PLot error curve
#plt.plot(train_err2, label='training_curve')
#plt.plot(test_err, label='testing_curve')
#legend = plt.legend(loc='upper right', shadow= True)
#plt.show()

# Plot accuracy curve
plt.plot(train_accuracy, label='training_curve')
plt.plot(test_accuracy, label='testing_curve')
legend = plt.legend(loc='upper right', shadow= True)
plt.show()

print('best test classification accuracy', np.max(test_accuracy))
np.set_printoptions(precision=4)
```

## 2.Multi-class classification

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

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].reshape(-1, 784) / 255.
    Target = Target[randIndx]
    Target_mat = np.zeros([len(Data),10])
```

```
    Target_mat[np.arange(len(Data)),Target] = 1
    Target = Target_mat
    trainData, trainTarget = Data[:15000], Target[:15000]
    validData, validTarget = Data[15000:16000], Target[15000:16000]
    testData, testTarget = Data[16000:], Target[16000:]

sess = tf.InteractiveSession()


lamda = 0.01
N = 0.05
batch_size = 500

number_updates = []
train_err1=[]
train_err2=[]
train_err3=[]
train_accuracy = []
test_err = []
test_accuracy = []

def buildGraph(lamda,n):
    # Variable creation
    W = tf.Variable(tf.truncated_normal(shape=[784,10]), name='weights')
#    b = tf.Variable(tf.truncated_normal(shape=[1,10]), name='biases')
    b = tf.Variable(0.0, name='biases')
    X = tf.placeholder(tf.float64, [None,784], name='input_x')
    y_target = tf.placeholder(tf.float64, [None,10], name='target_y')

    W = tf.cast(W, tf.float64)
    b = tf.cast(b, tf.float64)
    # Graph definition
    y_logits = tf.matmul(X, W) + b
#    y_logits = tf.reshape(y_logits, [-1,])
#    y_logits = tf.reshape(tf.matmul(X, W) + b, [-1,1,1])
    y_predicted = tf.nn.softmax(y_logits)
#    y_logits = tf.expand_dims(y_logits,[-1,])
    # Error definition
    softmaxCrossEntropyError =
tf.reduce_mean(tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y_logits,y_target),
name='softmaxCroEntroError') + lamda * tf.reduce_mean(tf.matmul(tf.transpose(W),W))/2, name='mean_error')
#    softmaxCrossEntropyError = tf.reduce_mean(tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y_logits,
y_target), reduction_indices=1, name='softmaxCroEntroError') + lamda *
tf.reduce_mean(tf.matmul(tf.transpose(W),W))/2, name='mean_error')
    # Training mechanism
    optimizer = tf.train.AdamOptimizer(learning_rate = n)
    train = optimizer.minimize(loss=softmaxCrossEntropyError)
    return W, b, X, y_target, y_logits, y_predicted, softmaxCrossEntropyError, train


def ClassAccuracy(Data, Target, currentW, currentb):
    y_predicted_v = tf.nn.softmax(tf.matmul(Data,currentW) + currentb)
    correct = tf.equal(tf.argmax(y_predicted_v,1),tf.argmax(Target,1))
    accuracy = tf.reduce_mean(tf.cast(correct,tf.float64)).eval()

    return accuracy
```

```
def runMult(n):
    train_err = []
    test_err = []
    #Build computation graph
    W, b, X, y_target, y_logits, y_predicted, softmaxCrossEntropyError, train = buildGraph(lamda,n)

    #Initialize session
    init = tf.initialize_all_variables()
    sess.run(init)
    # Training model
    for step in range(0,30):

        idx = np.arange(0,15000)
        np.random.shuffle(idx)
        for index in range(0,30):

            batch1 = trainData[idx]
            batch2 = trainTarget[idx]
            batch1 = batch1[index*batch_size:(index+1)*batch_size]
            batch2 = batch2[index*batch_size:(index+1)*batch_size]
            _, err, currentW, currentb, yhat_x, yhat = sess.run([train, softmaxCrossEntropyError, W, b, y_logits,
y_predicted], feed_dict={X: batch1, y_target: batch2})

            if (1) :

                print("Iter: %3d, ERR-train: %4.2f"%(int(15000/batch_size)*step+index, err))
                number_updates.append (int(15000/batch_size)*step+index)
                train_err.append (err)
                #train_accuracy.append (ClassAccuracy(batch1, batch2, currentW, currentb))
                # Testing model
                errTest = sess.run(softmaxCrossEntropyError, feed_dict= {X: testData, y_target: testTarget})
                print("Iter: %3d, ERR-test: %4.2f"%(int(15000/batch_size)*step+index, errTest))
                test_err.append (errTest)
                test_accuracy.append (ClassAccuracy(testData, testTarget, currentW, currentb))

    #plt.plot(number_updates, train_err, '-')
#    plt.plot(number_updates, test_err, '-')
#    plt.plot(number_updates, train_accuracy, '-')
#    plt.plot(number_updates, test_accuracy, '-')
    #plt.show()
    return train_err,test_err
#train_err1=runMult(0.001)
train_err2,test_err=runMult(0.01)
#train_err3=runMult(0.1)
#plt.plot(train_err1,label='learning rate = 0.001')
#plt.plot(train_err2,label='learning rate = 0.01')
#plt.plot(train_err3,label='learning rate = 0.1')
#le=plt.legend(loc='upper right', shadow=True)
#plt.show()

# Plot cross_entropy loss
#plt.plot(train_err2,label='training loss')
#plt.plot(test_err,label='testing loss')
#le=plt.legend(loc='upper right', shadow=True)
```

```
#plt.show()

# Plot accuracy
#plt.plot(train_accuracy,label='training accuracy')
#plt.plot(test_accuracy,label='testing accuracy')
#le=plt.legend(loc='lower right', shadow=True)
#plt.show()
print('best accuracy',np.max(test_accuracy))
np.set_printoptions(precision=4)
```

## 3. Neural network

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

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].reshape(-1, 784) / 255.
    Target = Target[randIndx]
    Target_mat = np.zeros([len(Data), 10])
    Target_mat[np.arange(len(Data)), Target] = 1
    Target = Target_mat
    trainData, trainTarget = Data[:15000], Target[:15000]
    validData, validTarget = Data[15000:16000], Target[15000:16000]
    testData, testTarget = Data[16000:], Target[16000:]

sess = tf.InteractiveSession()

n_nodes_hl1 = 1000
n_classes = 10
batch_size = 500

lamda = 0.0003

n_updates = []
train_err =[]
train_accuracy = []
test_err = []
test_accuracy = []
valid_err = []
valid_accuracy = []

x = tf.placeholder(tf.float64, [None, 784], name='input_x')
y = tf.placeholder(tf.float64, name='target_y')

def neural_network_model(d, nodes):

    hidden_layer_1 = {'weights': tf.Variable(tf.random_normal([784, nodes], stddev=3./(nodes+batch_size))),
                      'biases': tf.Variable(0.0, [nodes,])}
    output_layer = {'weights': tf.Variable(tf.random_normal([nodes, n_classes], stddev=3./(nodes+n_classes))),
                    'biases': tf.Variable(0.0, [n_classes,])}
```

```
    hidden_layer_1['weights'] = tf.cast(hidden_layer_1['weights'], tf.float64)
    hidden_layer_1['biases'] = tf.cast(hidden_layer_1['biases'], tf.float64)
    output_layer['weights'] = tf.cast(output_layer['weights'], tf.float64)
    output_layer['biases'] = tf.cast(output_layer['biases'], tf.float64)

    l1 = tf.add(tf.matmul(d, hidden_layer_1['weights']), hidden_layer_1['biases'])
    l1 = tf.nn.relu(l1)
    output = tf.matmul(l1, output_layer['weights']) + output_layer['biases']

    return output, hidden_layer_1['weights'], output_layer['weights']


# neural_network_model(testData, nodes)


def train_neural_network(x,n):

    prediction, W1, W = neural_network_model(x, n_nodes_hl1)
    cost = tf.reduce_mean(tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(prediction,y) +lamda *
(tf.reduce_mean(tf.square(W1)) + tf.reduce_mean(tf.square(W)))/2))
    optimizer = tf.train.AdamOptimizer(learning_rate=n).minimize(cost)

    number_epochs = 30

    init = tf.initialize_all_variables()
    sess.run(init)

    for epoch in range(number_epochs):

        idx = np.arange(0, len(trainData))
        np.random.shuffle(idx)
        for i in range(0, int(len(trainData) / batch_size)):
            batch1 = trainData[idx]
            batch2 = trainTarget[idx]
            batch1 = batch1[i * batch_size:(i + 1) * batch_size]
            batch2 = batch2[i * batch_size:(i + 1) * batch_size]
#           epoch_data = np.float64(epoch_data)
#           epoch_data = trainData.next_batch(batch_size)
#           epoch_target = trainTarget.next_batch(batch_size)
#           p, c, _ = sess.run([prediction, cost, optimizer], feed_dict = {x: batch1, y: batch2})

            _, c, p = sess.run([optimizer, cost, prediction], feed_dict={x: batch1, y: batch2})

            if i== 0:
                print('Updates', epoch * int(len(trainData) / batch_size) + i, 'completed out of', number_epochs *
int(len(trainData) / batch_size), 'loss:', c)

                train_err.append(c)
                n_updates.append(epoch*int(len(trainData) / batch_size)+i)

                correct = np.equal(np.argmax(p, 1), np.argmax(batch2, 1))
                accuracy = np.float64(1) - np.mean(np.float64(correct))
                train_accuracy.append(accuracy)


    #TestData
```

```
            test_e, p_test = sess.run([cost,prediction], feed_dict={x: testData, y: testTarget})
            test_err.append(test_e)
            correct_test = np.equal(np.argmax(p_test, 1), np.argmax(testTarget, 1))
            accuracy_test =np.float64(1) - np.mean(np.float64(correct_test))
            test_accuracy.append(accuracy_test)

    #ValidData
            valid_e, p_valid = sess.run([cost,prediction], feed_dict={x: validData, y: validTarget})
            valid_err.append(valid_e)
            correct_valid = np.equal(np.argmax(p_valid, 1), np.argmax(validTarget, 1))
            accuracy_valid =np.float64(1) - np.mean(np.float64(correct_valid))
            valid_accuracy.append(accuracy_valid)


#    print('Accuracy:',accuracy.eval({x:validData, y:validTarget}))
#    print('Accuracy:',accuracy.eval({x:testData, y:testTarget}))
    #plt.plot(n_updates, train_err, '-')
    #plt.plot(n_updates, test_err, '-')
    #plt.plot(n_updates, valid_err, '-')
#    plt.plot(train_accuracy)
#    plt.plot(test_accuracy)
#    plt.plot(valid_accuracy)
    #plt.show()
#train_err1 = []
#train_err2 = []
#train_err3 = []
#train_err1 = train_neural_network(x,0.01)
train_err2 = train_neural_network(x,0.001)
#train_err3 = train_neural_network(x,0.0001)
#plt.plot(train_err1,label='learning rate = 0.01')
#plt.plot(train_err2,label='learning rate = 0.001')
#plt.plot(train_err3,label='learning rate = 0.0001')
plt.plot(train_accuracy,label='train_err')
plt.plot(test_accuracy,label='test_err')
plt.plot(valid_accuracy,label='valid_err')
le=plt.legend(loc='upper right', shadow=True)
plt.show()
print(np.argmin(valid_accuracy))
print('valid',valid_accuracy[13])
print('test',test_accuracy[13])
print('train',train_accuracy[13])
```

## 4.neural network (with dropout)

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

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].reshape(-1, 784) / 255.
    Target = Target[randIndx]
```

```
    Target_mat = np.zeros([len(Data), 10])
    Target_mat[np.arange(len(Data)), Target] = 1
    Target = Target_mat
    trainData, trainTarget = Data[:15000], Target[:15000]
    validData, validTarget = Data[15000:16000], Target[15000:16000]
    testData, testTarget = Data[16000:], Target[16000:]

sess = tf.InteractiveSession()

n_nodes_hl1 = 1000
n_classes = 10
batch_size = 500
N = 0.9
lamda = 0.0003

n_updates = []
train_err = []
train_accuracy = []
test_err = []
test_accuracy = []
valid_err = []
valid_accuracy = []

x = tf.placeholder(tf.float64, [None, 784], name='input_x')
y = tf.placeholder(tf.float64, name='target_y')
keep_prob = tf.placeholder(tf.float64)

def neural_network_model(d, nodes, dropout):

    hidden_layer_1 = {'weights': tf.Variable(tf.random_normal([784, nodes], stddev=3./(nodes+batch_size))),
                  'biases': tf.Variable(0.0, [nodes,])}
    output_layer = {'weights': tf.Variable(tf.random_normal([nodes, n_classes], stddev=3./(nodes+n_classes))),
                 'biases': tf.Variable(0.0, [n_classes,])}

    hidden_layer_1['weights'] = tf.cast(hidden_layer_1['weights'], tf.float64)
    hidden_layer_1['biases'] = tf.cast(hidden_layer_1['biases'], tf.float64)
    output_layer['weights'] = tf.cast(output_layer['weights'], tf.float64)
    output_layer['biases'] = tf.cast(output_layer['biases'], tf.float64)

    l1 = tf.add(tf.matmul(d, hidden_layer_1['weights']), hidden_layer_1['biases'])
    l1 = tf.nn.relu(l1)

    l1 = tf.nn.dropout(l1, dropout)

    output = tf.matmul(l1, output_layer['weights']) + output_layer['biases']

    return output, hidden_layer_1['weights'], output_layer['weights']


# neural_network_model(testData, nodes)


def train_neural_network(x):
    prediction, W1, W = neural_network_model(x, n_nodes_hl1, keep_prob)
    cost = tf.reduce_mean(tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(prediction,y) + lamda *
(tf.reduce_mean(tf.square(W1)) + tf.reduce_mean(tf.square(W)))/2))
```

```
    optimizer = tf.train.AdamOptimizer().minimize(cost)

    number_epochs = 60

    init = tf.initialize_all_variables()
    sess.run(init)

    for epoch in range(number_epochs):

        idx = np.arange(0, len(trainData))
        np.random.shuffle(idx)
        for i in range(0, int(len(trainData) / batch_size)):
            batch1 = trainData[idx]
            batch2 = trainTarget[idx]
            batch1 = batch1[i * batch_size:(i + 1) * batch_size]
            batch2 = batch2[i * batch_size:(i + 1) * batch_size]
#            epoch_data = np.float64(epoch_data)
#            epoch_data = trainData.next_batch(batch_size)
#            epoch_target = trainTarget.next_batch(batch_size)
#            p, c, _ = sess.run([prediction, cost, optimizer], feed_dict = {x: batch1, y: batch2})

            _, c, p = sess.run([optimizer, cost, prediction], feed_dict={x: batch1, y: batch2, keep_prob: 0.5})
            print('Updates', epoch * int(len(trainData) / batch_size) + i, 'completed out of', number_epochs *
int(len(trainData) / batch_size), 'loss:', c)
            if i == 0:
                train_err.append(c)
                n_updates.append(epoch*int(len(trainData) / batch_size)+i)

                correct = np.equal(np.argmax(p, 1), np.argmax(batch2, 1))
                accuracy = np.float64(1) - np.mean(np.float64(correct))
                train_accuracy.append(accuracy)

            #TestData
            #test_e, p_test = sess.run([cost,prediction], feed_dict={x: testData, y: testTarget, keep_prob: 1})
            #test_err.append(test_e)
            #correct_test = np.equal(np.argmax(p_test, 1), np.argmax(testTarget, 1))
            #accuracy_test = np.mean(np.float64(correct_test))
            #test_accuracy.append(accuracy_test)

            #ValidData
                valid_e, p_valid = sess.run([cost,prediction], feed_dict={x: validData, y: validTarget, keep_prob: 1})
                valid_err.append(valid_e)
                correct_valid = np.equal(np.argmax(p_valid, 1), np.argmax(validTarget, 1))
                accuracy_valid = np.float64(1) - np.mean(np.float64(correct_valid))
                valid_accuracy.append(accuracy_valid)


#    print('Accuracy:',accuracy.eval({x:validData, y:validTarget}))
#    print('Accuracy:',accuracy.eval({x:testData, y:testTarget}))
#    plt.plot(n_updates, train_err, '-')
#    plt.plot(n_updates, test_err, '-')
#    plt.plot(n_updates, valid_err, '-')
train_neural_network(x)
plt.plot(train_accuracy, label='train')
    # plt.plot(test_accuracy, label='test')
plt.plot(valid_accuracy, label='valid')
```

```
legend = plt.legend(loc='upper right', shadow=True)
plt.show()
print('training errors = ', train_accuracy[59])
print('validation errors =', valid_accuracy[59])
```