CSE253 Programming Assignment 1

Chao Yu
PID:A99049546
chy018@eng.ucsd.edu

Chao Long
PID:A53224755
c3long@eng.ucsd.edu

Abstract

In this assignment we applied Logistic and Softmax Regression to classify handwritten digits from Yann LeCun's website for the MNIST Database. We use batch gradient descent and mini-batch gradient descent for the model. In addition,we tuned the hyper-parameters such as the learning rate, batchsize and regularizer. In order to prevent over-fitting for the training model, we observed the result of hold-out set and decide the number of epochs we are going to use. Furthermore, we compare L1 and L2 regularization to penalize the complexity of the model. We use L2 regularization with mini-batch gradient descent to achieve 96.82 percent correctness on test set. We do the same thing using Softmax to train 0-9 digits classifier and combining with L2 regularization and mini-batch gradient descent to get 93.50 percent correctness on test set.

1 Introduction

Logistic Regression and Softmax Regression are two powerful machine learning methods to do classification or recognition job. In the assignment, we are using Logistic Regression and Softmax Regression to make classification of MNIST data. In the training process, we apply mini-batch to accelerate the training time and use regularization, early stop to prevent over fitting. In addition, we compare the effects of L1 and L2 regularization and choose the best regularization factor λ for the assignment.

2 Logistic Regression

2.1 Data Pre-processing

We pick the first 20000 data as our training data set and last 2000 data as test set. Then we choose the data with label 2 and 3 in order to train a 2-3 Logistic Regression model and choose data with label 2 and 8 to train a 2-8 Logistic Regression model.

After extracting the dataset we want, we shuffle the training data and use 90 percent of training data as the real training data,the left 10 percent of training data to help us tune hyperparameters.

Then we normalize the data and add "1" to each data to represent bias. The weights are the number of features for each data plus one bias. Initialize weights with mean equals 0 and standard deviation equals 0.01.

2.2 Derive the gradient for Logistic Regression

Because the equation $g_w(x) = \frac{1}{1+exp(-w^T x)}$ is a sigmoid function, derivate of a sigmoid function g' is g(x)(1-g(x)).

From the equation:

$$E\left(w\right) = -\frac{1}{N}\sum_{n=1}^{N}\left\{t^n lny^n + \left(1-t^n\right)ln\left(1-y^n\right)\right\} \tag{1}$$

we take the derivative of $E^n\left(w\right)$ to get dw:

$$\begin{aligned}
\frac{\partial E^n\left(w\right)}{\partial w_{jk}} &= \frac{-\partial\left(t^n lny^n + \left(1-t^n\right)ln\left(1-y^n\right)\right)}{\partial w_{jk}} \\
&= -\frac{\partial\left(t^n lny^n + \left(1-t^n\right)ln\left(1-y^n\right)\right)}{\partial y^n}\frac{\partial y^n}{\partial w_{jk}} \\
&= -\frac{\partial\left(t^n lny^n + \left(1-t^n\right)ln\left(1-y^n\right)\right)}{\partial y^n}\frac{\partial g_w(x)}{\partial w_{jk}} \\
&= -\frac{\partial\left(t^n lny^n + \left(1-t^n\right)ln\left(1-y^n\right)\right)}{\partial y^n}\frac{\partial}{\partial w_{jk}}\frac{1}{1+exp\left(-w^T x\right)} \\
&= -\left(\frac{t^n}{y^n} - \frac{1-t^n}{1-y^n}\right)y^n\left(1-y^n\right)x_j \\
&= -\left(t^n\left(1-y^n\right) - \left(1-t^n\right)y^n\right)x_j \\
&= -\left(t^n - y^n\right)x_j
\end{aligned} \tag{2}$$

2.3 Training

We use minibatch to train our model and pick the mini batch size 256. For 2-3 data sets we have 3600 training datasets and 405 validation sets. The training data is divided to 15 mini-batches. For each epoch, we go though the 15 epochs to update gradients of weights. We choose the our initial learning rate to be 0.005 and applying the learning rate decay equation:

$$\eta\left(t\right) = \eta\left(0\right)/\left(1 + t/T\right) \tag{3}$$

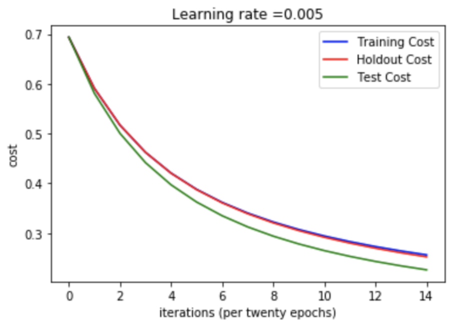
And we pick T=1000 to make sure our learning rate will not drop too fast. For early stopping, if the error on the hold-out set begins to go up consistently 5 epochs, we will break the training process.

2.4 Results

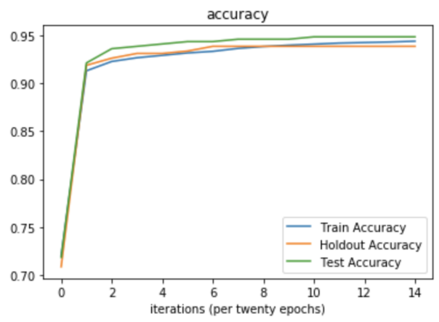
We plot the cost of the training set, validation set and test set along with the accuracies using batch gradient descent in Figure 1(a), 1(b) and mini-batch gradient descent in 2(a), 2(b). Comparing with 1(a), 1(b) and 2(a), 2(b), we observe the cost using mini-batch drops faster than using one batch. For mini-batch gradient descent, the accuracy of the training set reaching 96.49 percent, validation set accuracy reaches 95.58 and the test accuracy reaches 96.82 percent. For batch gradient descent, the accuracy of the training set reaching 95.84 percent, validation set accuracy reaches 95.23 percent and the test accuracy reaches 96.12 percent. In real world we will not be able to see the test data set, so we compare the performances between batch gradient descent and mini-batch gradient descent by comparing the accuracy of validation set. We can see that accuracy of validation set by mini-batch is higher than using one batch, so the result of using mini-batch is better. In addition, in mini-batch gradient descent, we update the weights for each mini-batch and update the weights 15 times for each epoch(we have 15 mini-batches total), the weights change more smoothly.

2.5 Discussion

The result of the cost between training set, validation set and test set are not as we expect. The validation set does not work well as a good stand-in for the test set. In 1(a), 1(b) and 2(a), 2(b), the cost of the test set is slightly lower than the validation set and the accuracy of training set is higher than the validation set, which are reasonable because the validation set and training set are from the same distribution. However, the cost of the test is the lowest and the accuracy of the test is the highest. We think the reason is that test data set between 2 and 3 are very easy to classify and test data set has different distributions with training set and validation set.

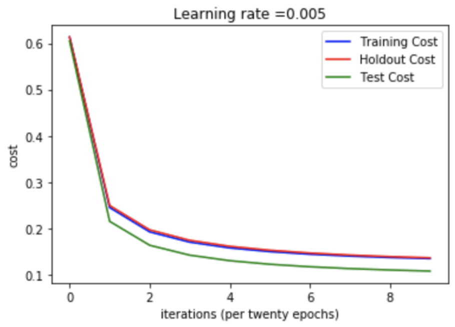


(a) $costerror$

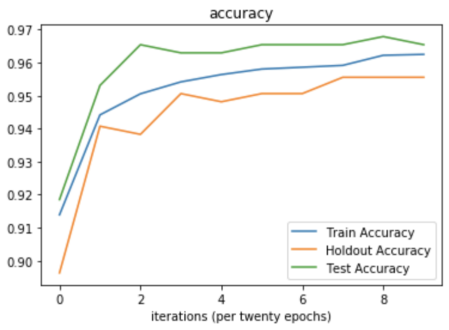


(b) $accuracy$

Figure 1: Performance of batch gradient descent with learning rate 0.005(2vs3)



(a) $costerror$



(b) $accuracy$

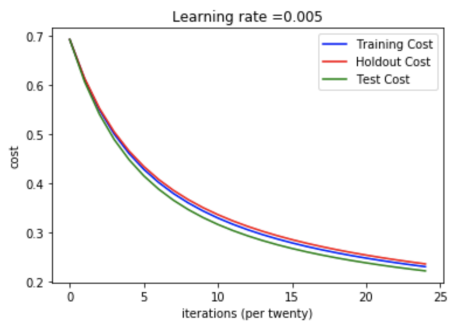
Figure 2: Performance of mini-batch gradient descent with learning rate 0.005(2vs3)

2.6 2 vs 8

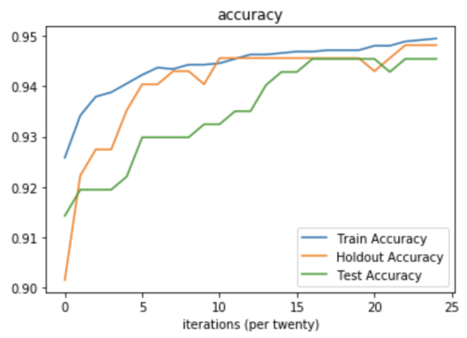
The data pre-processing and training process is the same as 2 vs 8. We choose the same learning rate = 0.005 and learning rate decay parameter T = 1000. Figure 3 demonstrates the cost and accuracy of training, validation and test set. We can see cost are almost the same between these three datasets. We achieve 94.95 percent accuracy in training set, 94.81 percent in validation set and 94.55 percent in test set. In Figure 4, we get 96.23 percent accuracy in training set, 95.87 percent in validation set and 96.12 percent in Test set. The perform of mini-batch gradient descent is slightly better than batch gradient descent by comparing the accuracy between validation set. For 2 and 8, the validation set can represent the test set because from the accuracy plot in both Figure 3 and Figure 4, validation set accuracy and test set accuracy are lower than the training set, indicating the distribution between validation set and test set are close to each other.

2.7 Weights difference

The weights for two classifiers are able to show what the classify look like. For example, in Figure 5, in 2 vs 3 case, we can see a "Three" shown dark. The weights difference between 2 vs 3 and 2 vs 8 is actually the weights of 3 vs 8, the 3,8 classifier, shown in Figure 6. The reason is the weight to represent classifier is actually a vector, and vector can be added or subtracted by Parallelogram Method. And the logistic regression is a linear classifier so that subtracting two classifiers can be a new classifier. We test the 3 vs 8 classifier by subtracting 2 vs 3 and 2 vs 8 classier and the new classifier perform well.

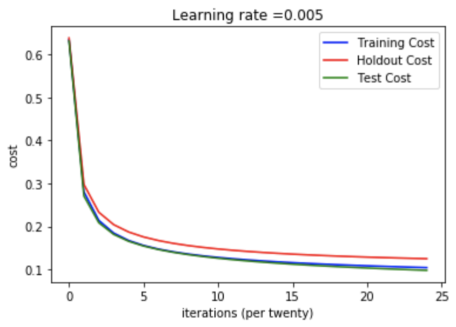


(a) $costerror$

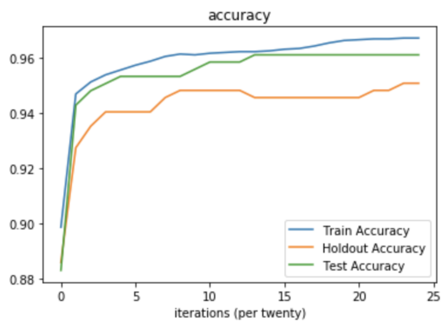


(b) $accuracy$

Figure 3: Performance of batch gradient descent with learning rate 0.005(2vs8)

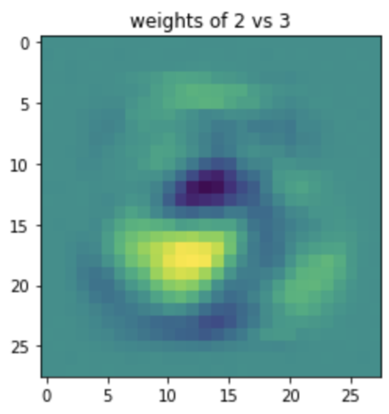


(a) $costerror$

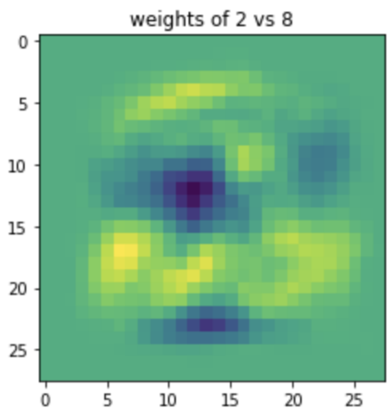


(b) $accuracy$

Figure 4: Performance of mini-batch gradient descent with learning rate 0.005(2vs8)



(a) $2vs3$



(b) $2vs8$

Figure 5: weights of 2vs3 and 2vs8

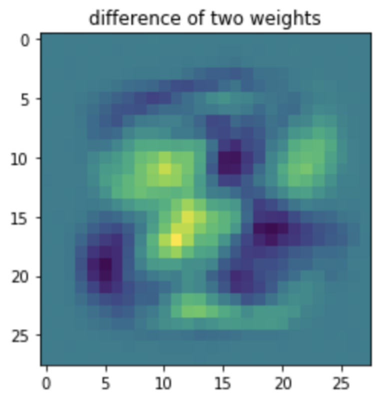


Figure 6: difference between two weights

3 Regularization

3.1 Method

We use L1 and L2 regularizations to train the data, the derivation of gradient to L1 and L2 can be shown below:
L1 Regularization:

$$\begin{aligned}\frac{\partial J}{\partial w_j} &= \frac{\partial E}{\partial w_j} + \lambda \frac{\partial C}{\partial w_j} \\ &= \frac{\partial E}{\partial w_j} + \lambda \frac{\partial}{\partial w_j} \sum_{i=1}^{N} |w_i| \\ &= \frac{\partial E}{\partial w_j} + \lambda \quad if \quad w_j > 0 \\ &= \frac{\partial E}{\partial w_j} - \lambda \quad if \quad w_j < 0\end{aligned} \tag{4}$$

L2 Regularization

$$\begin{aligned}\frac{\partial J}{\partial w_j} &= \frac{\partial E}{\partial w_j} + \lambda \frac{\partial C}{\partial w_j} \\ &= \frac{\partial E}{\partial w_j} + \lambda \frac{\partial}{\partial w_j} \sum_{i=1} w_i^2 \\ &= \frac{\partial E}{\partial w_j} + 2\lambda w_j\end{aligned} \tag{5}$$

3.2 Result for L1 regularization

For L1 regularization, we choose different value of λ form 10^{-6} to 1. From experiments we see the length of the weight vector tends to be 0 as λ increases. This is reasonable according to the general result that L1 regularization easily leads weight to become sparse during optimization. In my point of view, the reason is that the constrain of L1 regularization $\sum |w_i|$ = 0 has sharp corner, leading its optimized weight always lies in axis, thus the weight vector has more 0 value in different dimension, leading to sparse image.

We train the logistic regression model with L1 regularization, and then we plot the percent correctness of training set over these different value of λ. The scale of x axis is $\ln \lambda$ as shown Figure 7(a) . Then we plot the length of weight vector over different λ , as shown in Figure 8(a). Then we explore the impact of λ to test set, as shown in Figure 9(a). Finally, we plot the weights vector of L1 regularization in each case as a image shown in Figure 10(a).

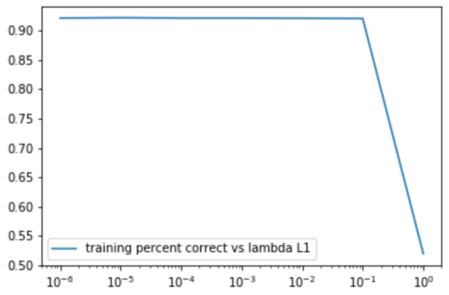
3.3 Result for L2 regularization

We train logistic regression model by L2 regularization with different value of λ. The general procedure is almost the same as L1 regularization shown above. Firstly, we plot the percent correctness of training set over different value of λ in Figure 7(b). We can see that as λ increases, the correctness is decreasing, which is reasonable according to regularized cost function. Concretely, Small λ, leading to small penalty to cost function, has tiny impact on the cost function. However, large λ will dominant the cost function, thus the model becomes not much optimal.

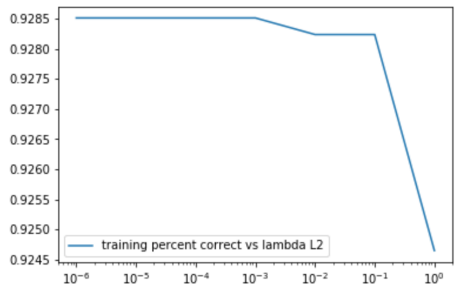
We plot the length of the weight vector for each λ, as shown in Figure 8(b). As λ increases, the length of the weight vector decrease, and this is reasonable according to cost function. Because large value of λ will create large decay to weight vector during training. And then we start to consider the impact of different value of λ on the testing data. As we can see in Figure 9(b) below. The percent correctness will increase to a peak and then goes down. This is reasonable according to the regularized model. Concretely, small λ has tiny impact on the cost function and it also suffers from overfitting as before. However, large λ makes the model becomes simple since the regularized part will dominant, leading to underfitting. So the optimal value of λ should be in the middle, like 10^{-4} in the figure. Finally, we plot the weight of different case as an image, showing in Figure 10(b).

3.4 Discussion for L1 and L2 regularization

We can see as λ increases, the weight images in L1 regularization tends to be sparse. And an optimal choice of λ is in the middle to form a good classifier like it is shown in Figure$10(a)$. Due to this property, L1 regularization can be used for feature selection by changing the weight of different dimension of w_i. However, L2 regularization leads all dimension of weight to become smaller at the same time because of its circular contour constrain and we can also choose an optimal λ just like Figure$10(b)$.

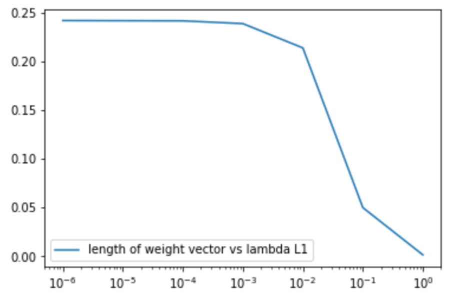


(a) $L1 \quad regularization$

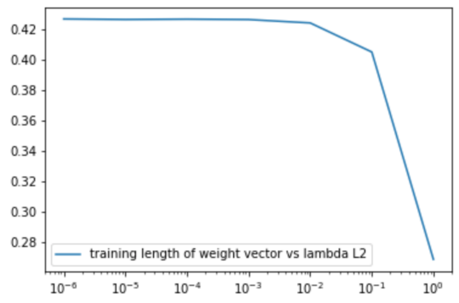


(b) $L2 \quad regularization$

Figure 7: Training percent correct over different value of λ

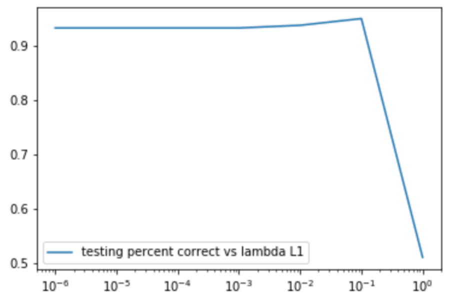


(a) $L1 \quad regularization$

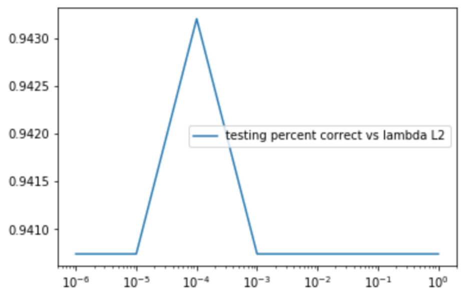


(b) $L2 \quad regularization$

Figure 8: Length of weight vector over different value of λ

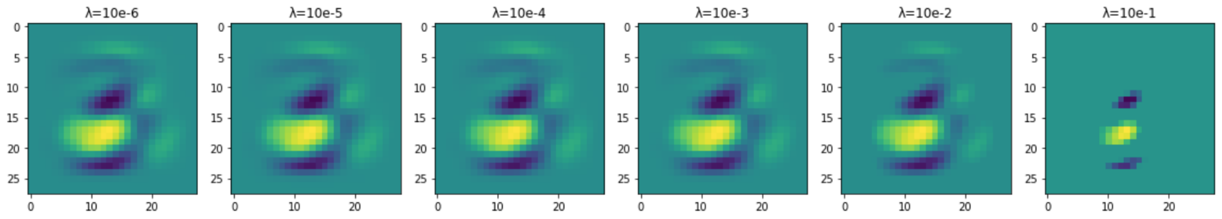


(a) $L1 \quad regularization$

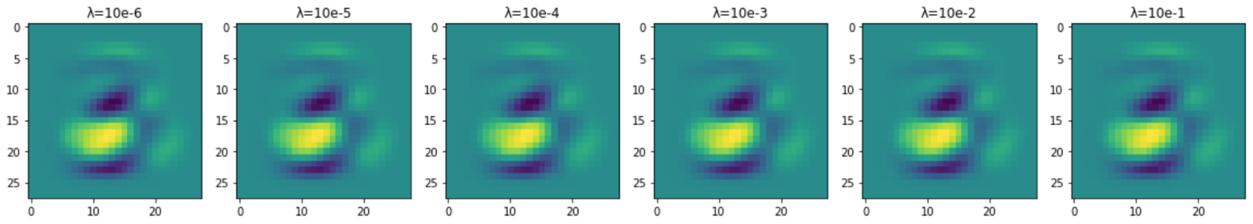


(b) $L2 \quad regularization$

Figure 9: Testing percent correct over different value of λ



(a) $L1 \quad weight \quad images$



(b) $L2 \quad weight \quad images$

Figure 10: Weight vector images over different value of λ

4 Softmax Regression via Gradient Descent

4.1 Derive the gradient for Softmax Regression

Assuming $\quad t^n_{k^l} = 1, \quad by \quad$ defination

$$\begin{aligned}
\nabla = -\frac{\partial E^n(w)}{\partial w_{jk}} \\
&= \quad -\frac{\partial}{\partial y^n_{k^l}}\left\{-t^n_{k^l}\cdot \ln y^n_{k^l}\right\}\cdot\frac{\partial y^n_{k^l}}{\partial a^n_k}\cdot\frac{\partial a^n_k}{\partial w_{jk}} \\
&= \quad \left(\frac{t^n_{k^l}}{y^n_{k^l}}\right)\cdot\frac{\partial}{\partial a^n_k}\left\{\frac{exp\left(a^n_{k^l}\right)}{\sum_{k'} exp\left(a^n_{k'}\right)}\right\}\cdot\frac{\partial}{\partial w_{jk}}\left\{w^T_k\cdot x^n\right\} \\
&= \quad \left(\frac{t^n_{k^l}}{y^n_{k^l}}\right)\cdot y^n_{k^l}\left(\delta_{kk^l}-y^n_k\right)\cdot x^n_j, \quad when \quad k=k^l,\delta_{kk^l}=1; \quad k\neq k^l,\delta_{kk^l}=0
\end{aligned} \tag{6}$$

$when \quad k=k^l, \quad t^n_k = t^n_{k^l} = 1$

$when \quad k\neq k, \quad t^n_k = 0 \quad , \quad which \quad is \quad the \quad same \quad as \quad \delta_{kk^l}$

$so \quad \nabla = \left(t^n_k - y^n_k\right)\cdot x^n_j$

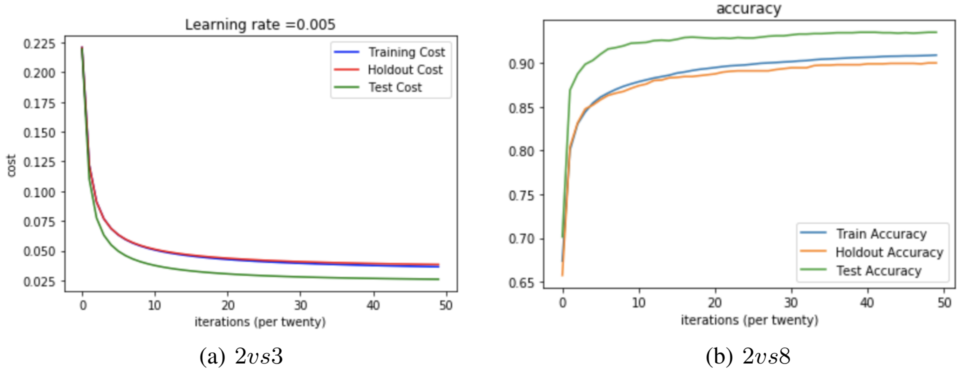
4.2 Data pre-processing and training

We use the similar data pre-processing and training precess. We use first 20000 data as training set and last 2000 to be test set. Then shuffle the training data and pick 90 percent data to be training data and the rest 10 percent data to be validation set. We applied mini-batch gradient descent with mini-batch size to be 1024 and early stopping epoch to be 5.

4.3 Result

In real world we will not be able to see the test set, so we reported our best results when the accuracy of the validation set is the highest.

After comparing different learning rate, regularization factor λ and L1, L2 regularization equation, we finally choose the best result when learning rate to be 0.005, λ to be 0.0001 and L2 regularization equation. In Figure 11,we get 90.89 percent accuracy for training set, 90.00 percent accuracy for validation set and 93.50 accuracy for test set. The accuracy of validation set is lower than the training set while the accuracy of test set is higher than the training set, showing the distribution of validation set is different from the test set. With 90.00 percent accuracy for validation set is the highest we are able to get.



(a) $2vs3$ (b) $2vs8$

Figure 11: minibatch for softmax

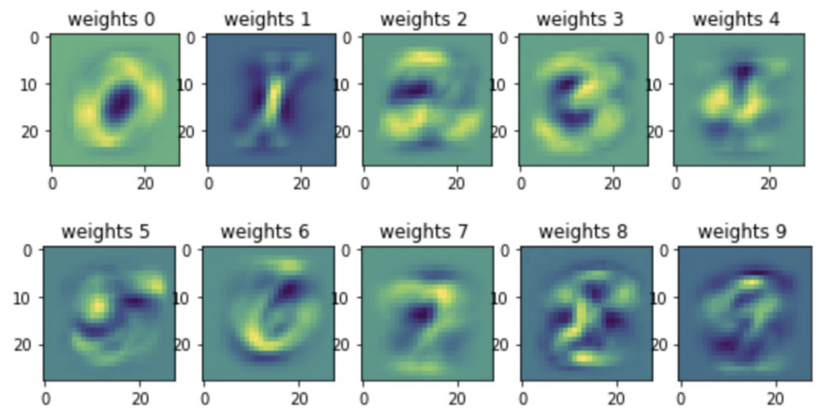
We created an image of the weights for each digit and compared average of all examples of this digit in Figure 12. We observed that the weights for each digit are very similar to the average of this digit. The more similar for each pairs are, the better the Softmax model will be.

5 Summary

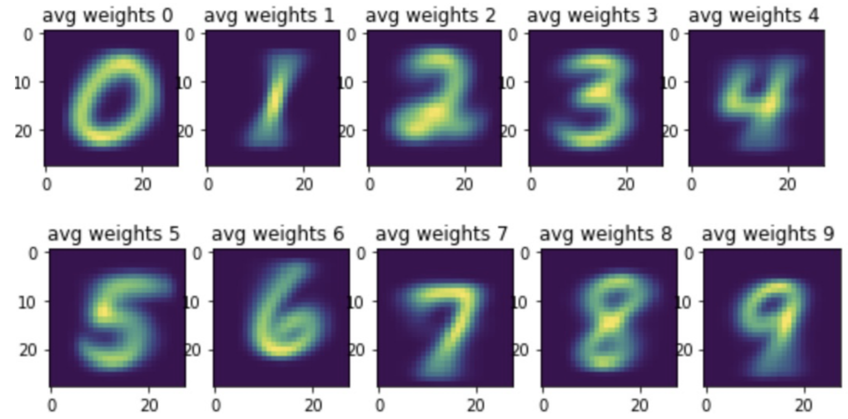
In this assignment, we learn to build shallow neuron network step by step based on logistic regression and softmax regression. Firstly, we start to build a simple model by matrix implementation, computing loss, getting gradient and checking the percent of correctness. Then we try to use several techniques to get more optimal model, including mini-batch gradient decent, L1 and L2 regularization, annel learning rate and early stopping. After accomplishing this assignment, we have a basic understanding of low level neuron network. And this is very useful for the following studying of high level deep learning algorithm.

6 Contribution

We do coding separately for verification. In general, Chao Yu focusing more on logistic regression and softmax regression. Chao Long is responsible more on equation derivation and regularization part.



(a) $weightsfrom0to9$



(b) $averageweightsoftrainingsets$

Figure 12: average weights of training data

7 Appendix

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math
from mnist import MNIST
mndata = MNIST('./mnistdata')
Train_images, Train_labels = mndata.load_training()
Test_images, Test_labels = mndata.load_testing()
Train_X = np.array(Train_images[0:20000]).T
Train_Y = np.array(Train_labels[0:20000],ndmin=2)
Test_X = np.array(Test_images[-2000:]).T
Test_Y = np.array(Test_labels[-2000:],ndmin=2)
#extract 2 and 3 from training data
Train=np.concatenate((Train_X,Train_Y),axis=0)
df=pd.DataFrame(Train.T)
last_column = df.columns[-1]
df.rename(columns={last_column:'label'}, inplace=True)
index_2s = df['label'] == 2
index_3s = df['label'] == 3
Train_2_3 = df.loc[index_2s | index_3s]
Train_2_3_X=Train_2_3.drop(['label'],axis=1).as_matrix()
```

```
Train_2_3_Y=Train_2_3['label'].as_matrix()
Train_2_3_X=Train_2_3_X.T
#extract 2 and 3 from test data
Test=np.concatenate((Test_X,Test_Y),axis=0)
df_test=pd.DataFrame(Test.T)
last_column = df_test.columns[-1]
df_test.rename(columns={last_column:'label'}, inplace=True)
index_2s = df_test['label'] == 2
index_3s = df_test['label'] == 3
Test_2_3 = df_test.loc[index_2s | index_3s]
Test_2_3_X=Test_2_3.drop(['label'],axis=1).as_matrix()
Test_2_3_Y=Test_2_3['label'].as_matrix()
Test_2_3_X=Test_2_3_X.T
Train_2_3_Y=abs(Train_2_3_Y-3) ##change 2 to label 1, change 3 to label 0
Test_2_3_Y=abs(Test_2_3_Y-3)
# add bias to train and divide 255
Train_2_3_X=Train_2_3_X/255.0
Test_2_3_X=Test_2_3_X/255.0
one=np.ones((1,Train_2_3_X.shape[1]))
Train_2_3_X=np.concatenate((one,Train_2_3_X),axis=0)
one=np.ones((1,Test_2_3_X.shape[1]))
Test_2_3_X=np.concatenate((one,Test_2_3_X),axis=0)
#shuffle data
shuffle_index = list(np.random.permutation(Train_2_3_X.shape[1]))
Train_2_3_X = Train_2_3_X[:, shuffle_index]
Train_2_3_Y = Train_2_3_Y[shuffle_index]
##add hold out set
Train_2_3_X_hold = Train_2_3_X[:,Train_2_3_X.shape[1]/10*9:]
Train_2_3_X = Train_2_3_X[:,:Train_2_3_X.shape[1]/10*9]
Train_2_3_Y_hold = Train_2_3_Y[len(Train_2_3_Y)/10*9:]
Train_2_3_Y = Train_2_3_Y[:len(Train_2_3_Y)/10*9]
#w=np.random.randn(Train_2_3_X.shape[0],1)*0.001 # initialize weights
w=np.zeros((Train_2_3_X.shape[0],1))
Train_2_3_Y.reshape((1,len(Train_2_3_Y)))
Test_2_3_Y.reshape((1,len(Test_2_3_Y)))
print Train_2_3_X.shape , Train_2_3_X_hold.shape,Train_2_3_Y.shape,Train_2_3_Y_hold.
def sigmoid(x):
    return 1.0/(1.0+np.exp(-x))

def cal_wx(w,x):
    return np.dot(w.T,x)
def cross_entropy(w,x,y):
    t=sigmoid(cal_wx(w,x)) # calculate activation
    cost_E= -1.0/(x.shape[1]) * np.sum(y*np.log(t)+(1-y)*np.log(1-t)) #calculate cos
    dw = 1.0/(x.shape[1])*np.dot((y-t),x.T)
    return cost_E,dw
def mini_batches(x, y, mini_batch_size = 256):

    m = x.shape[1]
    mini_batches = []

    #calculate number of complete batches
    num_complete_minibatches = m//mini_batch_size
    for k in range(0, num_complete_minibatches):
        mini_batch_x = x[:,k * mini_batch_size:(k + 1) * mini_batch_size]
        mini_batch_y = y[k * mini_batch_size:(k + 1) * mini_batch_size]
        mini_batch = (mini_batch_x, mini_batch_y)
        mini_batches.append(mini_batch)
    if m % mini_batch_size != 0:
        end = m - mini_batch_size * math.floor(m / mini_batch_size)
        mini_batch_x = x[:,num_complete_minibatches * mini_batch_size:]
        mini_batch_y = y[num_complete_minibatches * mini_batch_size:]
        mini_batch = (mini_batch_x, mini_batch_y)
        mini_batches.append(mini_batch)

    return mini_batches
w=np.random.randn(Train_2_3_X.shape[0],1)*0.001 # initialize weights
def optimize_reg(w, x, y, x_hold,y_hold, x_test,y_test,num_iterations, learning_rate
    costs = []
    costs_hold = []
    costs_test = []
    accuracy_train = []
    accuracy_hold = []
    accuracy_test = []
    for i in range(num_iterations):
        # Cost and gradient calculation (add regualarization)
        cost_E,dw=cross_entropy(w,x,y)
        cost_E+=0.5*reg*np.sum(w**2)  # add regularization
        dw+=reg*w.T

        #cost for hold out
        cost_E_hold,_=cross_entropy(w,x_hold,y_hold)
        cost_E_hold += 0.5*reg*np.sum(abs(w))

        #cost for test
        cost_E_test,_=cross_entropy(w,x_test,y_test)
        cost_E_test += 0.5*reg*np.sum(abs(w))

        #update weights
        w = w+learning_rate*dw.T
        learning_rate/=(1+i/1000) #learning rate decay

        #calculate accuracy
        _, acc_train = prediction(w,x,y)
        _, acc_hold = prediction(w,x_hold,y_hold)
        _, acc_test = prediction(w,x_test,y_test)



        # Record the costs and acc
        if i % 20 == 0:
            costs.append(cost_E)
            costs_hold.append(cost_E_hold)
            costs_test.append(cost_E_test)
            accuracy_train.append(acc_train)
            accuracy_hold.append(acc_hold)
            accuracy_test.append(acc_test)
            print "Train␣Cost␣after␣iteration␣%i:␣%f" %(i, cost_E)
            print "Accuracy␣for␣train␣is␣", acc_train
           # print "Accuracy for hold is ", acc_hold
            #print "Accuracy for test is ", acc_test


    return  w,costs,costs_hold,costs_test,accuracy_train,accuracy_hold,accuracy_test

def prediction(w,x,y):
    prediction=sigmoid(cal_wx(w,x))
    predict=np.zeros_like(prediction)
    for index,number in enumerate(prediction[0]):
        if number>=0.5:
            predict[0,index]=1
        else:
            predict[0,index]=0
    #calculate accuracy
    correct = [(a==b) for (a,b) in zip(predict[0],y)]
    acc = sum(correct) * 1.0 / len(correct)
    return predict , acc
w=np.random.randn(Train_2_3_X.shape[0],1)*0.001 # initialize weights

def mini_optimize_reg(w, x, y, x_hold,y_hold, x_test,y_test,num_epochs, learning_rate
    costs = []
    costs_hold = []
    costs_test = []
    accuracy_train = []
    accuracy_hold = []
    accuracy_test = []
    cost_epoch = []
    for i in range(num_epochs):

        minibatches = mini_batches(x, y, mini_batch_size )

        for minibatch in minibatches:
            (minibatch_x, minibatch_y) = minibatch
            _,dw=cross_entropy(w,minibatch_x,minibatch_y)
            #cost_E+=0.5*reg*np.sum(w**2)  # add regularization
            dw += reg*w.T

            #update weights
            w = w+learning_rate*dw.T
            learning_rate/=(1+i/1000) #learning rate decay


        # Cost for train
        cost_E,_ = cross_entropy(w,x,y)
        cost_E += 0.5*reg*np.sum(w**2)


        #cost for hold out
        cost_E_hold,_=cross_entropy(w,x_hold,y_hold)
        cost_E_hold += 0.5*reg*np.sum(w**2)

        #cost for test
        cost_E_test,_=cross_entropy(w,x_test,y_test)
        cost_E_test += 0.5*reg*np.sum(w**2)

        #early stop

        if i > 5:
            if cost_epoch[-5:] == sorted(cost_epoch[-5:]):
            break
        cost_epoch.append(cost_E_hold)
        #calculate accuracy
        _, acc_train = prediction(w,x,y)
        _, acc_hold = prediction(w,x_hold,y_hold)
        _, acc_test = prediction(w,x_test,y_test)
        # Record the costs and acc
        if i % 20 == 0:
            costs.append(cost_E)
            costs_hold.append(cost_E_hold)
            costs_test.append(cost_E_test)
            accuracy_train.append(acc_train)
            accuracy_hold.append(acc_hold)
            accuracy_test.append(acc_test)
            print "Train␣Cost␣after␣iteration␣%i:␣%f" %(i, cost_E)
            print "Accuracy␣for␣train␣is␣", acc_train
           # print "Accuracy for hold is ", acc_hold
            #print "Accuracy for test is ", acc_test


    return  w,costs,costs_hold,costs_test,accuracy_train,accuracy_hold,accuracy_test
w,costs,costs_hold,costs_test,accuracy_train,accuracy_hold,accuracy_test=mini_optimi
w,costs,costs_hold,costs_test,accuracy_train,accuracy_hold,accuracy_test = optimize_
costs = np.squeeze(costs)
costs_hold = np.squeeze(costs_hold)
costs_test = np.squeeze(costs_test)
plt.plot(costs,color = 'blue',label='Training␣Cost')
plt.plot(costs_hold,color = 'red',label='Holdout␣Cost')
plt.plot(costs_test,color = 'green',label='Test␣Cost')
plt.ylabel('cost')
plt.xlabel('iterations␣(per␣twenty␣epochs)')
plt.title("Learning␣rate␣=" + str(0.005))
plt.legend()
plt.show()
plt.plot(accuracy_train, label='Train␣Accuracy')
plt.plot(accuracy_hold, label='Holdout␣Accuracy')
plt.plot(accuracy_test,label='Test␣Accuracy')
plt.xlabel('iterations␣(per␣twenty␣epochs)')
plt.title("accuracy")
plt.legend()
plt.show()
wp=w[1:].reshape((28,28))
plt.imshow(wp)
plt.title("weights␣of␣2␣vs␣3")
plt.show()
#2 and 8
Train_X = np.array(Train_images[0:20000]).T
Train_Y = np.array(Train_labels[0:20000],ndmin=2)
Test_X = np.array(Test_images[-2000:]).T
Test_Y = np.array(Test_labels[-2000:],ndmin=2)
#extract 2 and 8 from training data
Train=np.concatenate((Train_X,Train_Y),axis=0)
df=pd.DataFrame(Train.T)
last_column = df.columns[-1]
df.rename(columns={last_column:'label'}, inplace=True)
index_2s = df['label'] == 2
index_8s = df['label'] == 8
Train_2_8 = df.loc[index_2s | index_8s]
Train_2_8_X=Train_2_8.drop(['label'],axis=1).as_matrix()
Train_2_8_Y=Train_2_8['label'].as_matrix()
Train_2_8_X=Train_2_8_X.T
#extract 2 and 8 from test data
Test=np.concatenate((Test_X,Test_Y),axis=0)
df_test=pd.DataFrame(Test.T)
```

```
last_column = df_test.columns[-1]
df_test.rename(columns={last_column:'label'}, inplace=True)
index_2s = df_test['label'] == 2
index_8s = df_test['label'] == 8
Test_2_8 = df_test.loc[index_2s | index_8s]
Test_2_8_X=Test_2_8.drop(['label'],axis=1).as_matrix()
Test_2_8_Y=Test_2_8['label'].as_matrix()
Test_2_8_X=Test_2_8_X.T
Train_2_8_Y=abs(Train_2_8_Y-8)/6 ##change 2 to label 1, change 3 to label 0
Test_2_8_Y=abs(Test_2_8_Y-8)/6
# add bias to train and divide 255
Train_2_8_X=Train_2_8_X/255.0
Test_2_8_X=Test_2_8_X/255.0
one=np.ones((1,Train_2_8_X.shape[1]))
Train_2_8_X=np.concatenate((one,Train_2_8_X),axis=0)
one=np.ones((1,Test_2_8_X.shape[1]))
Test_2_8_X=np.concatenate((one,Test_2_8_X),axis=0)
#shuffle data
shuffle_index = list(np.random.permutation(Train_2_8_X.shape[1]))
Train_2_8_X = Train_2_8_X[:, shuffle_index]
Train_2_8_Y = Train_2_8_Y[shuffle_index]
##add hold out set
Train_2_8_X_hold = Train_2_8_X[:,Train_2_8_X.shape[1]/10*9:]
Train_2_8_X = Train_2_8_X[:,:Train_2_8_X.shape[1]/10*9]
Train_2_8_Y_hold = Train_2_8_Y[len(Train_2_8_Y)/10*9:]
Train_2_8_Y = Train_2_8_Y[:len(Train_2_8)/10*9]
Train_2_8_Y.reshape((1,len(Train_2_8_Y)))
Test_2_8_Y.reshape((1,len(Test_2_8_Y)))
#w=np.random.randn(Train_2_8_X.shape[0],1)*0.0001 # initialize weights
w=np.zeros((Train_2_8_X.shape[0],1))
print Train_2_8_X.shape , Train_2_8_X_hold.shape,Train_2_8_Y.shape,Train_2_8_Y_hold.
w,costs,costs_hold,costs_test,accuracy_train,accuracy_hold,accuracy_test = mini_opti
w,costs,costs_hold,costs_test,accuracy_train,accuracy_hold,accuracy_test = optimize_
costs = np.squeeze(costs)
costs_hold = np.squeeze(costs_hold)
costs_test = np.squeeze(costs_test)
costs_test = np.squeeze(costs_test)
plt.plot(costs,color = 'blue',label='Training␣Cost')
plt.plot(costs_hold,color = 'red',label='Holdout␣Cost')
plt.plot(costs_test,color = 'green',label='Test␣Cost')
plt.ylabel('cost')
plt.xlabel('iterations␣(per␣twenty)')
plt.title("Learning␣rate␣=" + str(0.005))
plt.legend()
plt.show()
plt.plot(accuracy_train, label='Train␣Accuracy')
plt.plot(accuracy_hold, label='Holdout␣Accuracy')
plt.plot(accuracy_test,label='Test␣Accuracy')
plt.xlabel('iterations␣(per␣twenty)')
plt.title("accuracy")
plt.legend()
plt.show()
wp8=w[1:].reshape((28,28))
plt.imshow(wp8)
plt.title("weights␣of␣2␣vs␣8")
plt.show()
wp28=wp-wp8
plt.imshow(wp28)
plt.title("difference␣of␣two␣weights")
plt.show()
#softmax
Train_X = np.array(Train_images[0:20000]).T
Train_Y = np.array(Train_labels[0:20000])
Test_X = np.array(Test_images[-2000:]).T
Test_Y = np.array(Test_labels[-2000:])
# add bias to train and divide 255
Train_X=Train_X/255.0
Test_X=Test_X/255.0
one=np.ones((1,Train_X.shape[1]))
Train_X=np.concatenate((one,Train_X),axis=0)
one=np.ones((1,Test_X.shape[1]))
Test_X=np.concatenate((one,Test_X),axis=0)
#shuffle data
shuffle_index = list(np.random.permutation(Train_X.shape[1]))
Train_X = Train_X[:, shuffle_index]
Train_Y = Train_Y[shuffle_index]
##add hold out set
Train_X_hold = Train_X[:,Train_X.shape[1]/10*9:]
Train_X = Train_X[:,:Train_X.shape[1]/10*9]
Train_Y_hold = Train_Y[len(Train_Y)/10*9:]
Train_Y = Train_Y[:len(Train_Y)/10*9]
w=np.random.randn(Train_X.shape[0],10)*0.001 # initialize weights
Train_Y.reshape((1,len(Train_Y)))
Test_Y.reshape((1,len(Test_Y)))
print Train_X.shape , Train_X_hold.shape,Train_Y.shape,Train_Y_hold.shape
def softmax(a):
    a-=np.max(a,axis=0,keepdims=True) #stabilize
    sum_a=np.sum(np.exp(a),axis=0,keepdims=True)
    y_k= np.exp(a)/sum_a
    return y_k

def build_mask(y):

    mask=np.zeros((10,len(y)))
    mask[y,np.arange(len(y))] = 1

    return mask

def cross_entro_soft(x,y,w):

    a= np.dot(w.T,x)
    mask = build_mask(y)
    y_k = softmax(a)
    E = -np.sum(np.log(y_k[y,np.arange(x.shape[1])]))/x.shape[1]/10


    dw= 1.0*np.dot((mask-y_k),x.T)/x.shape[1]

    return E,dw

def predict_softmax(w,x,y):
    a= np.dot(w.T,x)
    y_k = softmax(a)
    y_pred = y_k.argmax(axis=0)
    correct = [(a==b) for (a,b) in zip(y_pred,y)]
    acc = sum(correct) * 1.0 / len(correct)
    return acc
    def optimize_reg_softmax(w, x, y,x_hold,y_hold,x_test,y_test, num_iterations, le
    costs = []
    costs_hold = []
    costs_test = []
    accuracy_train = []
    accuracy_hold = []
    accuracy_test = []

    for i in range(num_iterations):

        # Cost and gradient calculation (add regualarization)
        cost_E ,dw=cross_entro_soft(x,y,w)
        cost_E+=0.5*reg*np.sum(w**2)
        dw+=reg*w.T

        #cost for hold out
        cost_E_hold ,_=cross_entro_soft(x_hold ,y_hold ,w)
        cost_E_hold += 0.5*reg*np.sum(w**2)

        #cost for test
        cost_E_test ,_=cross_entro_soft(x_test ,y_test ,w)
        cost_E_test += 0.5*reg*np.sum(w**2)

        #calculate accuracy
        acc_train = predict_softmax(w,x,y)
        acc_hold = predict_softmax(w, x_hold , y_hold)
        acc_test = predict_softmax(w, x_test , y_test)

        #update weights
        w = w+learning_rate*dw.T

        #learning rate decay
        learning_rate/=(1+i/1000)

        # Record the costs
        if i % 100 == 0:
            costs.append(cost_E)
            costs_hold.append(cost_E_hold)
            costs_test.append(cost_E_test)
            accuracy_train.append(acc_train)
            accuracy_hold.append(acc_hold)
            accuracy_test.append(acc_test)

            print ("Cost␣after␣iteration␣%i:␣%f" %(i, cost_E))

    return w,costs ,costs_hold ,costs_test ,accuracy_train ,accuracy_hold ,accuracy_test
w=np.random.randn(Train_X.shape[0],10)*0.001 # initialize weights

def mini_optimize_reg_softmax(w, x, y, x_hold ,y_hold , x_test ,y_test ,num_epochs, lear
    costs = []
    costs_hold = []
    costs_test = []
    accuracy_train = []
    accuracy_hold = []
    accuracy_test = []
    cost_epoch = []
    for i in range(num_epochs):

        minibatches = mini_batches(x, y, mini_batch_size )
        for minibatch in minibatches:
            (minibatch_x, minibatch_y) = minibatch
            _,dw=cross_entro_soft(minibatch_x,minibatch_y,w)
            #cost_E+=0.5*reg*np.sum(w**2)  # add regularization
            dw += reg*w.T

            #updata gradients and learning rate decay
            w = w+learning_rate*dw.T
            learning_rate/=(1+i/1000) #learning rate decay


        # Cost for train
        cost_E,_ = cross_entro_soft(x,y,w)
        cost_E += 0.5*reg*np.sum(w**2) #L2
        #cost_E += 0.5*reg*np.sum(abs(w)) #L1


        #cost for hold out
        cost_E_hold,_=cross_entro_soft(x_hold,y_hold,w)
        cost_E_hold += 0.5*reg*np.sum(w**2) #L2
        #cost_E_hold += 0.5*reg*np.sum(abs(w)) #L1

        #cost for test
        cost_E_test,_=cross_entro_soft(x_test,y_test,w)
        cost_E_test += 0.5*reg*np.sum(w**2) #L2
        #cost_E_test += 0.5*reg*np.sum(abs(w)) #L1

        #calculate accuracy
        acc_train = predict_softmax(w,x,y)
        acc_hold = predict_softmax(w,x_hold,y_hold)
        acc_test = predict_softmax(w,x_test,y_test)

        cost_epoch.append(cost_E_hold)

        #early stop
        if i > 5:
            if cost_epoch[-5:] == sorted(cost_epoch[-5:]):
                break


        # Record the costs and acc
        if i % 20 == 0:
            costs.append(cost_E)
            costs_hold.append(cost_E_hold)
            costs_test.append(cost_E_test)
            accuracy_train.append(acc_train)
            accuracy_hold.append(acc_hold)
            accuracy_test.append(acc_test)
            print "Train␣Cost␣after␣iteration␣%i:␣%f" %(i, cost_E)
            print "Accuracy␣for␣train␣is␣", acc_train
           # print "Accuracy for hold is ", acc_hold
            #print "Accuracy for test is ", acc_test


    return w,costs,costs_hold,costs_test,accuracy_train,accuracy_hold,accuracy_test
w,costs,costs_hold,costs_test,accuracy_train,accuracy_hold,accuracy_test=mini_optimi
costs = np.squeeze(costs)
costs_hold = np.squeeze(costs_hold)
costs_test = np.squeeze(costs_test)
```

```
plt.plot(costs,color = 'blue',label='Training␣Cost')
plt.plot(costs_hold,color = 'red',label='Holdout␣Cost')
plt.plot(costs_test,color = 'green',label='Test␣Cost')
plt.ylabel('cost')
plt.xlabel('iterations␣(per␣twenty)')
plt.title("Learning␣rate␣=" + str(0.005))
plt.legend()

plt.show()
plt.plot(accuracy_train, label='Train␣Accuracy')
plt.plot(accuracy_hold, label='Holdout␣Accuracy')
plt.plot(accuracy_test,label='Test␣Accuracy')
plt.xlabel('iterations␣(per␣twenty)')
plt.title("accuracy")
plt.legend()
plt.show()
print accuracy_train[-1]
print accuracy_hold[-1]
print accuracy_test[-1]
fig = plt.figure(figsize=(9, 9))
plt.subplot(2, 5, 1)
plt.imshow(w[:,0][1:].reshape((28,28)))
plt.title("weights␣0␣")
plt.subplot(2, 5, 2)
plt.imshow(w[:,1][1:].reshape((28,28)))
plt.title("weights␣1␣")
plt.subplot(2, 5, 3)
plt.imshow(w[:,2][1:].reshape((28,28)))
plt.title("weights␣2␣")
plt.subplot(2, 5, 4)
plt.imshow(w[:,3][1:].reshape((28,28)))
plt.title("weights␣3␣")
plt.subplot(2, 5, 5)
plt.imshow(w[:,4][1:].reshape((28,28)))
plt.title("weights␣4␣")
plt.subplot(2, 5, 6)
plt.imshow(w[:,5][1:].reshape((28,28)))
plt.title("weights␣5␣")
plt.subplot(2, 5, 7)
plt.imshow(w[:,6][1:].reshape((28,28)))
plt.title("weights␣6␣")
plt.subplot(2, 5, 8)
plt.imshow(w[:,7][1:].reshape((28,28)))
plt.title("weights␣7␣")
plt.subplot(2, 5, 9)
plt.imshow(w[:,8][1:].reshape((28,28)))
plt.title("weights␣8␣")
plt.subplot(2, 5, 10)
plt.imshow(w[:,9][1:].reshape((28,28)))
plt.title("weights␣9␣")
plt.show()
Train_X = np.array(Train_images[0:20000]).T
Train_Y = np.array(Train_labels[0:20000],ndmin=2)
Train=np.concatenate((Train_X,Train_Y),axis=0)
df=pd.DataFrame(Train.T)
last_column = df.columns[-1]
df.rename(columns={last_column:'label'}, inplace=True)
index_0s = df['label'] == 0
Train_0 = df.loc[index_0s]
```

```
Train_0_X=Train_0.drop(['label'],axis=1).as_matrix()
Train_0_X=Train_0_X.T
index_1s = df['label'] == 1
Train_1 = df.loc[index_1s]
Train_1_X=Train_1.drop(['label'],axis=1).as_matrix()
Train_1_X=Train_1_X.T
index_2s = df['label'] == 2
Train_2 = df.loc[index_2s]
Train_2_X=Train_2.drop(['label'],axis=1).as_matrix()
Train_2_X=Train_2_X.T
index_3s = df['label'] == 3
Train_3 = df.loc[index_3s]
Train_3_X=Train_3.drop(['label'],axis=1).as_matrix()
Train_3_X=Train_3_X.T
index_4s = df['label'] == 4
Train_4 = df.loc[index_4s]
Train_4_X=Train_4.drop(['label'],axis=1).as_matrix()
Train_4_X=Train_4_X.T
index_5s = df['label'] == 5
Train_5 = df.loc[index_5s]
Train_5_X=Train_5.drop(['label'],axis=1).as_matrix()
Train_5_X=Train_5_X.T
index_6s = df['label'] == 6
Train_6 = df.loc[index_6s]
Train_6_X=Train_6.drop(['label'],axis=1).as_matrix()
Train_6_X=Train_6_X.T
index_7s = df['label'] == 7
Train_7 = df.loc[index_7s]
Train_7_X=Train_7.drop(['label'],axis=1).as_matrix()
Train_7_X=Train_7_X.T
index_8s = df['label'] == 8
Train_8 = df.loc[index_8s]
Train_8_X=Train_8.drop(['label'],axis=1).as_matrix()
Train_8_X=Train_8_X.T
index_9s = df['label'] == 9
Train_9 = df.loc[index_9s]
Train_9_X=Train_9.drop(['label'],axis=1).as_matrix()
Train_9_X=Train_9_X.T
Train_0_X=Train_0_X/255.0
Train_0_X_average=np.sum(Train_0_X,axis=1)/Train_1_X.shape[1]
Train_1_X=Train_1_X/255.0
Train_1_X_average=np.sum(Train_1_X,axis=1)/Train_1_X.shape[1]
Train_2_X=Train_2_X/255.0
Train_2_X_average=np.sum(Train_2_X,axis=1)/Train_2_X.shape[1]
Train_3_X=Train_3_X/255.0
Train_3_X_average=np.sum(Train_3_X,axis=1)/Train_3_X.shape[1]
Train_4_X=Train_4_X/255.0
Train_4_X_average=np.sum(Train_4_X,axis=1)/Train_4_X.shape[1]
Train_5_X=Train_5_X/255.0
Train_5_X_average=np.sum(Train_5_X,axis=1)/Train_5_X.shape[1]
Train_6_X=Train_6_X/255.0
Train_6_X_average=np.sum(Train_6_X,axis=1)/Train_6_X.shape[1]
Train_7_X=Train_7_X/255.0
Train_7_X_average=np.sum(Train_7_X,axis=1)/Train_7_X.shape[1]
Train_8_X=Train_8_X/255.0
Train_8_X_average=np.sum(Train_8_X,axis=1)/Train_8_X.shape[1]
Train_9_X=Train_9_X/255.0
Train_9_X_average=np.sum(Train_9_X,axis=1)/Train_9_X.shape[1]
fig = plt.figure(figsize=(9, 9))
```

```
plt.subplot(2, 5, 1)
plt.imshow(Train_0_X_average.reshape((28,28)))
plt.title("avg␣weights␣0␣")
plt.subplot(2, 5, 2)
plt.imshow(Train_1_X_average.reshape((28,28)))
plt.title("avg␣weights␣1␣")
plt.subplot(2, 5, 3)
plt.imshow(Train_2_X_average.reshape((28,28)))
plt.title("avg␣weights␣2␣")
plt.subplot(2, 5, 4)
plt.imshow(Train_3_X_average.reshape((28,28)))
plt.title("avg␣weights␣3␣")
plt.subplot(2, 5, 5)
plt.imshow(Train_4_X_average.reshape((28,28)))
plt.title("avg␣weights␣4␣")
plt.subplot(2, 5, 6)
plt.imshow(Train_5_X_average.reshape((28,28)))
plt.title("avg␣weights␣5␣")
plt.subplot(2, 5, 7)
plt.imshow(Train_6_X_average.reshape((28,28)))
plt.title("avg␣weights␣6␣")
plt.subplot(2, 5, 8)
plt.imshow(Train_7_X_average.reshape((28,28)))
plt.title("avg␣weights␣7␣")
plt.subplot(2, 5, 9)
plt.imshow(Train_8_X_average.reshape((28,28)))
plt.title("avg␣weights␣8␣")
plt.subplot(2, 5, 10)
plt.imshow(Train_9_X_average.reshape((28,28)))
plt.title("avg␣weights␣9␣")
plt.show()
#minibatch
def training_minibatch_L1(X_training,Y_training,X_holdout,Y_holdout,X_testing,Y_testi
    E_costs_training = []
    E_costs_holdout = []
    E_costs_testing = []
    accuracies_training = []
    accuracies_holdout = []
    accuracies_testing = []
    w,w0 = initialize(784)
    m = X_training.shape[1]
    weights = []
    weights_0 = []
    minibatch_size=256
    for epoch in range(num_epochs):
        epoch_cost = 0
        num_minibatches = int(m/minibatch_size)
        mini_batches = random_mini_batches(X_training,Y_training, 256)
        for j in range(len(mini_batches)):
            #(minibatch_X,minibatch_Y) = mini_batches[j]
            tmpgrads,E_cost_training_mini_batch = propagate_L1(w,w0,mini_batches[j][(
            E_cost_training = subcosting_L1(w,w0,X_training,Y_training,lambd)
            E_cost_holdout = subcosting_L1(w,w0,X_holdout,Y_holdout,lambd)
            E_cost_testing = subcosting_L1(w,w0,X_testing,Y_testing,lambd)
            accuracy_training = predict(w,w0,X_training,Y_training)
            accuracy_holdout = predict(w,w0,X_holdout,Y_holdout)
            accuracy_testing = predict(w,w0,X_testing,Y_testing)
            dw = tmpgrads["dw"]
            dw0 = tmpgrads["dw0"]
```

```
                if j == len(mini_batches)-1:
                    weights.append(np.array(w))
                    weights_0.append(np.array(w0))
                #params = {"w":w,"w0":w0}
                #grads  = {"dw":dw,"dw0":dw0}
                learning_rate = learning_rate_0/(1+0.001*(epoch+1))
                w = w - learning_rate*dw
                w0 = w0 - learning_rate*dw0
                #weights.append(np.array(w))
                #weights_0.append(np.array(w0))
                if j == len(mini_batches)-1:
                    E_costs_training.append(E_cost_training)
                    E_costs_holdout.append(E_cost_holdout)
                    E_costs_testing.append(E_cost_testing)
                    accuracies_training.append(accuracy_training)
                    accuracies_holdout.append(accuracy_holdout)
                    accuracies_testing.append(accuracy_testing)
            if epoch>1 and E_costs_holdout[epoch]>E_costs_holdout[epoch-1] and E_costs_h
                weights = weights[:-2]
                weights_0 = weights_0[:-2]
                E_costs_training = E_costs_training[:-2]
                E_costs_holdout  =  E_costs_holdout[:-2]
                E_costs_testing  =  E_costs_testing[:-2]
                accuracies_training  =  accuracies_training[:-2]
                accuracies_holdout = accuracies_holdout[:-2]
                accuracies_testing = accuracies_testing[:-2]
        return weights,weights_0,E_costs_training,E_costs_holdout,E_costs_testing,accura
# training_L1 for different lambd in one method
def total_training_L1():
    lambds = [0.000001,0.00001,0.0001,0.001,0.01,0.1,1]
    w_s_L1 = []
    accuracies_training_L1 = []
    accuracies_testing_L1 = []
    for i in range(len(lambds)):
        tmpweights_L1,tmpweights_0_L1,tmpcrossentropy_training_L1,tmpcrossentropy_hol
training_minibatch_L1(training_images_2_3,training_labels_2_3,holdout_images_2_3,hol
        w_s_L1.append(tmpweights_L1[-1])
        accuracies_training_L1.append(tmpaccuracies_training_L1[-1])
        accuracies_testing_L1.append(tmpccuracies_testing_L1[-1])

    return  w_s_L1,accuracies_training_L1,accuracies_testing_L1
#plotting L1: training vs lambd
def training_L1_lambd(accuracies_training_lambd_L1):
    lambds = [0.000001,0.00001,0.0001,0.001,0.01,0.1,1]
    fig,ax = plt.subplots()
    ax.plot(lambds,accuracies_training_lambd_L1,label="training␣percent␣correct␣vs␣la
    ax.set_xscale("log")
    ax.legend()
training_L1_lambd(accuracies_training_lambd_L1)
#plotting image of different weight vector by different value of lambda
def weights_L1_lambd_image(weight_lambd_L1):
    weight_graphs = []
    for i in range(len(weight_lambd_L1)):
        weight_graphs.append(weight_lambd_L1[i].reshape(28,28))
        #fig,ax =plt.subplots()
        #ax.imshow(weight_graphs)
    print(len(weight_lambd_L1))
    fig = plt.figure(figsize=(40, 20))
    ax1 = fig.add_subplot(1,6,1)
    ax1.set_title("lambd=10e-6")
    ax1.imshow(weight_graphs[0])
    ax2 = fig.add_subplot(1,6,2)
    ax2.set_title("lambd=10e-5")
    ax2.imshow(weight_graphs[1])
    ax3 = fig.add_subplot(1,6,3)
    ax3.set_title("lambd=10e-4")
    ax3.imshow(weight_graphs[2])

    ax4 = fig.add_subplot(1,6,4)
    ax4.set_title("lambd=10e-3")
    ax4.imshow(weight_graphs[3])
    ax5 = fig.add_subplot(1,6,5)
    ax5.set_title("lambd=10e-2")
    ax5.imshow(weight_graphs[4])
    ax6 = fig.add_subplot(1,6,6)
    ax6.set_title("lambd=10e-1")
    ax6.imshow(weight_graphs[5])
 weights_L1_lambd_image(weight_lambd_L1)
 #plotting L1: testing vs lambd
def testing_L1_lambd(accuracies_testing_lambd_L1):
    lambds = [0.000001,0.00001,0.0001,0.001,0.01,0.1,1]
    fig,ax = plt.subplots()
    ax.plot(lambds,accuracies_testing_lambd_L1,label="testing␣percent␣correct␣vs␣lamb
    ax.set_xscale("log")
    ax.legend()
    print(accuracies_testing_lambd_L1[-2])
weights_L1_lambd(weight_lambd_L1)
  testing_L1_lambd(accuracies_testing_lambd_L1)
```