Project 3

Handwritten Digit recognition using Logistic Regression, Neural Network and Convolution Neural Network

CSE474/574: Introduction to Machine Learning
(Fall 2016)

Instructor: Sargur N. Srihari

Teaching Assistants: Jun Chu, Junfei Wang and Kyung Won Lee

Sugosh Nagavara Ravindra

sugoshna@buffalo.edu

Person#: 50207357

Table of Contents

Abstract

Handwritten Digit recognition is a classification task and hence in this project we implement two basic algorithms, Logistic Regression, Neural Network and a deep learning algorithm Convolution Neural Network. We are using MNIST dataset and divide it as follows: 50,000 training set, 10,000 validation set, 10,000 testing set. We implement the algorithms in two fashions: Stochastic Gradient Descent and Mini-Batch Stochastic Gradient Descent.

We obtain varied results for all three algorithms. The optimal solution with right hyper-parameter tuning yielded **92.33%** testing accuracy on logistic regression, **97.38%** testing accuracy on Neural Network and **99.28%** testing accuracy on Convolution Neural Network.

We further use the pre-trained weights and try to predict USPS character dataset and prove the "No Free Lunch" theorem.

Introduction

In machine learning, classification is the problem of identifying to which of a set of categories a new observation belongs, on the basis of a training set of data containing observations whose category membership is known. Further in the report we look into each of the classification algorithms in detail. Since MNIST is a multiclass classification problem, we convert the output into vectored one-hot encoded format. This means that the output will be a vector of all zeros except in the position where the class is true, where it is set to one.

The input is a size of 784 (28x28), we retain these pixel values as features and train using all the models. No free lunch theorem basically states that if a model is extensively trained on one type of dataset, it won't have any effect of similar type of datasets. We test this theorem out using USPS dataset.

We look into Logistic Regression, Neural Network and Convolution neural network in detail in the following chapters. We then test out the "No free Lunch" theorem.

Logistic Regression

Logistic regression is used to classify data into two discrete output units like Pass/Fail, Positive/Negative. In this project, since we have 10 different classes we implement multiclass Logistic regression.

Multinomial logistic regression is a classification method that generalizes logistic regression to multiclass problems, i.e. with more than two possible discrete outcomes. That is, it is a model that is used to predict the probabilities of the different possible outcomes of a categorically distributed dependent variable, given a set of independent variables (which may be real-valued, binary-valued, categorical-valued, etc.).

If we had a binary classification problem, we could implement sigmoid activation function. But since we have a multiclass problem, we implement softmax activation function.

Softmax is represented by

$$p(\mathcal{C}_k|\mathrm{x}) = y_k(\mathrm{x}) = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$$

Where $a_k = \mathrm{w}_k^\top \mathrm{x} + b_k.$

b_k represents bias variable. We set the bias variable value as 1.

Before we start the training process, we need to convert out output from discrete value to 1-of-K encoding scheme, which means that we convert the output to a vectored form which is of dimension k, where k is number of classes and only one value is set to 1, which is the output class itself. It is basically a probabilistic representation of output. The one-hot encoding code is as follows:

```
def design_T(train_y,validation_y,test_y):
    T_train=np.zeros((len(train_y),10))
    T_validation=np.zeros((len(validation_y),10))
    T_test=np.zeros((len(test_y),10))
    for i in range(len(train_y)):
        T_train[i][train_y[i]]=1
    for i in range(len(validation_y)):
        T_validation[i][validation_y[i]]=1
    for i in range(len(test_y)):
        T_test[i][test_y[i]]=1
    return T_train,T_validation,T_test
```

We load the MNIST dataset using pickle. The next step is to divide the dataset, we divide it as follows: 50,000 training set, 10,000 validation set, 10,000 testing set. Following is the code for training.

```
for z in range(50):
    print "ITERATION "+str(z+1)
    for i in range(len(train_x)):
        A=np.dot(W,train_x[i])+1
        #Softmax Function
        denom = np.sum(np.exp(A))
        E=0
        for k in range(10):
            Y[k]=np.exp(A[k])/denom
            E-=np.dot(T_train[i][k],np.log(Y[k]))
        #cnt+=1
        for j in range(10):
            W[j]=W[j]-0.01*(Y[j]-T_train[i][j])*train_x[i]
        #    ETerm[j]+=((T_train[i][j]-Y[j])*train_x[i])
        #if(cnt%10==0):
        #    W=W+0.01*ETerm/10
        #    ETerm=np.zeros(W.shape)
    print E
```

In both logistic regression we minimize the cross entropy function

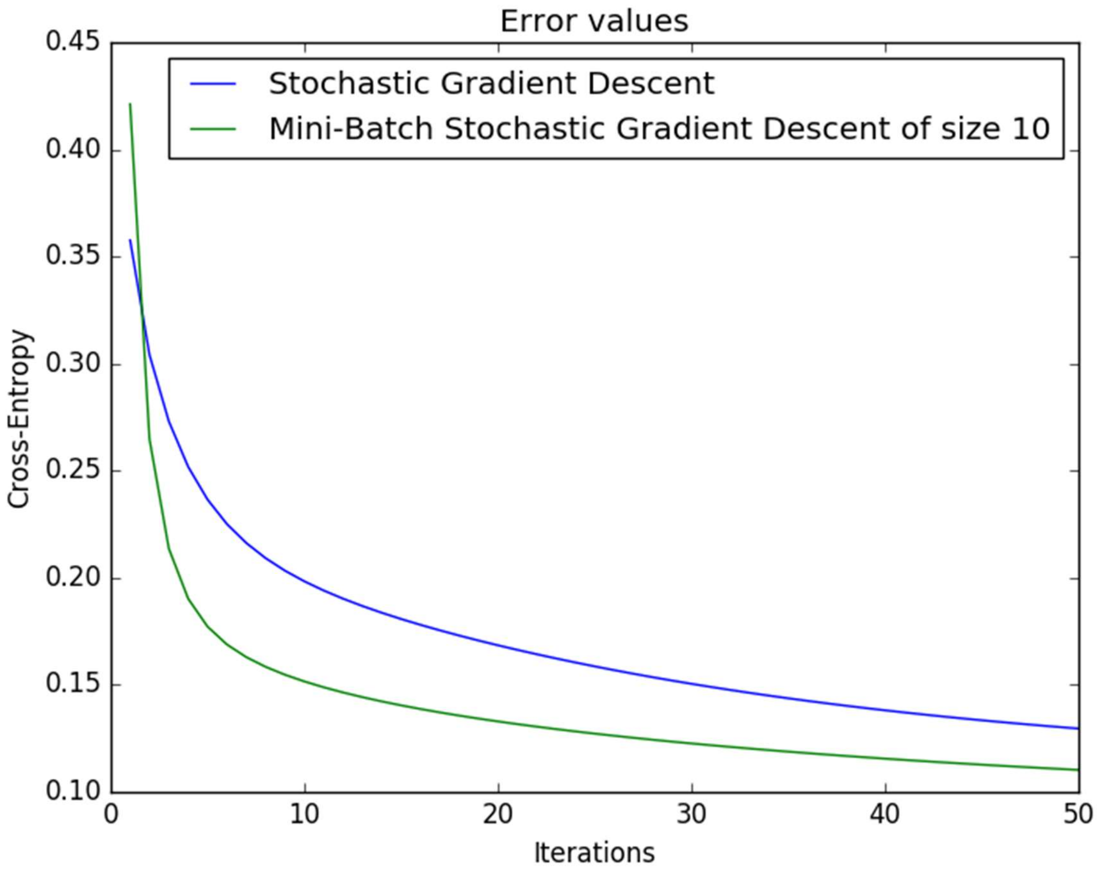
$$E(\mathrm{x}) = -\sum_{k=1}^{K} t_k \ln y_k$$

The gradient of the error term would be

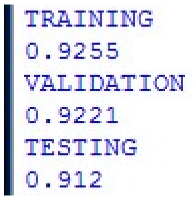
```
(Y[j]-T_train[i][j])*train_x[i]
```

Uncomment the above code to obtain solution for mini-batch stochastic gradient descent of batch size 10.
We set the initial value of weights to be all ones.
We couldn't iterate over all values of learning rate as it would cause overflow errors. From trial and error 0.01 was the optimal solution to learning rate.

Following is a graph obtained by plotting cross-entropy vs iterations.

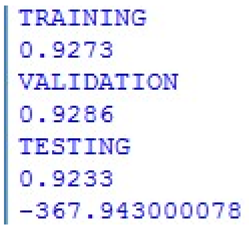


We obtained the following accuracy for 50 iterations:
Stochastic Gradient Descent:



```
TRAINING
0.9255
VALIDATION
0.9221
TESTING
0.912
```

The last line represents time taken.
Mini-batch stochastic gradient descent of batch size 10:

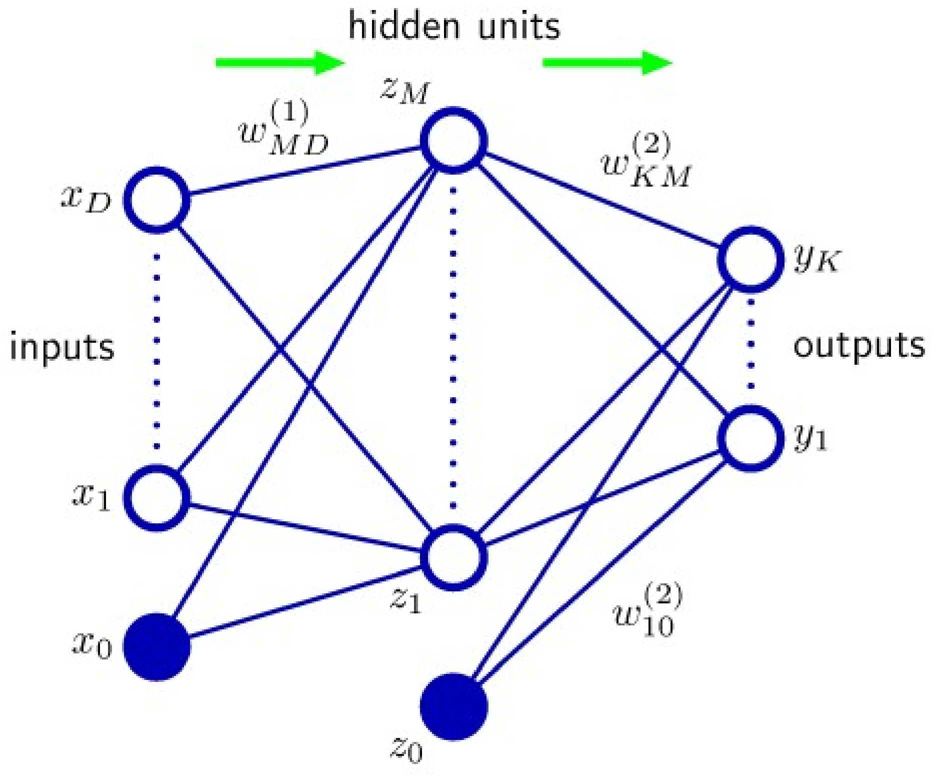


```
TRAINING
0.9273
VALIDATION
0.9286
TESTING
0.9233
-367.943000078
```

Neural Network

Neural networks are a computational approach which is based on a large collection of neural units loosely modeling the way a biological brain solves problems with large clusters of biological neurons connected by axons. This approach can be implemented for multiclass optimization problems. We look into how neural networks can be implemented to classify MNIST dataset.

Following is the visualization of a neural network



For this particular model we have 785 (784 + 1 bias) input features. We set the number of hidden units to be 100 as this provided the optimal result. Setting high number of hidden units can be effective but can be computationally very expensive as it is exponentially growing function in time and space.

We implement one of the three activation functions (Sigmoid / Tanh / Rectified Linear) for the hidden unit and Softmax function for the output layer.

Activation Function:

Following is the list of activation functions applied to the hidden unit for 5 iterations and using Stochastic Gradient Descent.

- Sigmoid activation function

$$S(t) = \frac{1}{1 + e^{-t}}$$

$$S'(t) = S(t)(1 - S(t))$$

```
TRAINING
0.96914
48457.0
VALIDATION
0.9673
9673.0
TESTING
0.962
9620.0
345.369999886
```

- Tanh activation function

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

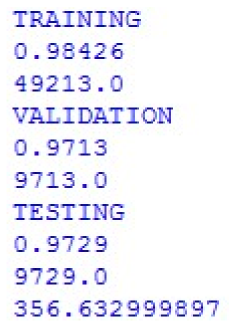
$$f(z) = 1 - (f(z))^2.)$$

```
TRAINING
0.98518
49259.0
VALIDATION
0.9714
9714.0
TESTING
0.9688
9688.0
470.134000063
```

- Rectified Linear unit

$y = \max(\epsilon, x)$ where epsilon is typically 0

$$\frac{dy}{dx} = \begin{cases} 1 & x > \epsilon \\ 0 & x \leq \epsilon \end{cases}$$



```
TRAINING
0.98426
49213.0
VALIDATION
0.9713
9713.0
TESTING
0.9729
9729.0
356.632999897
```

Clearly Rectified linear unit seems to be the best activation function for hidden units.

Codes for activation functions:

```
def activate(Z):
    #Sigmoid
    out=np.zeros(len(Z))
##    for i in range(len(Z)):
##        out[i]=1.0/(1.0+math.exp(-Z[i]))
##    #tanh
##    for i in range(len(Z)):
##        out[i]=(np.exp(2*Z[i])-1)/(np.exp(2*Z[i])+1)
##
##    #Relu
    for i in range(len(Z)):
        out[i]=max(0,Z[i])

    return out


def derivative(Z):
    out=np.zeros(Z.shape)
    #Sigmoid
##    out=Z*(1-Z)
    #tanh
##    out=(1-Z**2)
    #Relu
    for i in range(len(Z)):
        if(Z[i]>0):
            out[i]=1
    return out


def softmax(Z):
    out=np.zeros(len(Z))
    denom=np.sum(np.exp(Z))
    for i in range(len(Z)):
        out[i]=np.exp(Z[i])/denom
    return out
```

Back propagation:

Back propagation is a method of training artificial neural networks used in conjunction with an optimization method such as gradient descent.

Back propagation is the process of propagating error backward from the output layer to the input layer, by finding the gradients at each point and updating the weights cumulatively.

Following is the code for back propagation

```
W1=W1-0.01*np.asarray(np.asmatrix(delta_j[1:]).T*np.asmatrix(train_x[i]))
W2=W2-0.01*np.asarray(np.asmatrix(delta_k).T*np.asmatrix(A_new))
```

The full training code for neural network is as shown

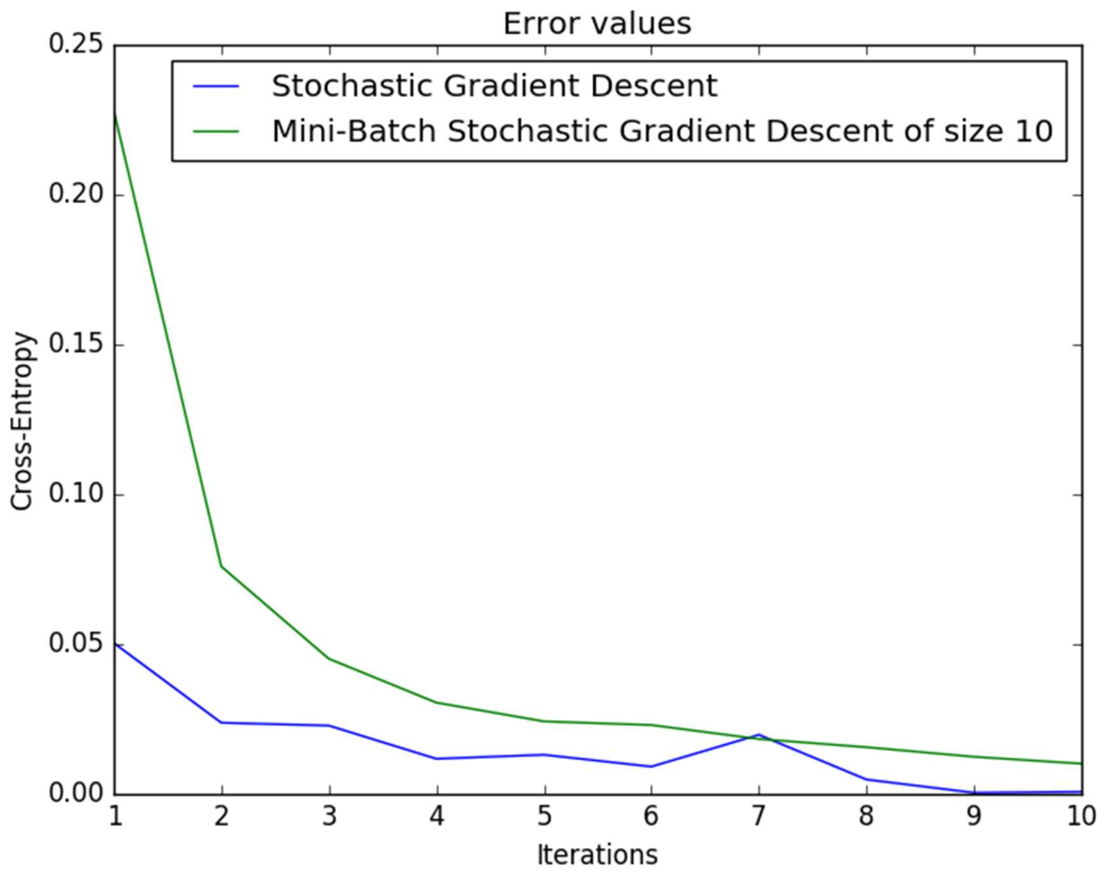
```
for k in range(5):
#    ETerm1=np.zeros(W1.shape)
#    ETerm2=np.zeros(W2.shape)
    print "Iteration ",k+1
    f.write(str(k+1)+" ")
    for i in range(len(train_x)):
        Z_2=np.dot(W1,train_x[i])
        A=activate(Z_2)
        A_new=np.append(1,A)
        Z_3=np.dot(W2,A_new)
        Y=softmax(Z_3)
        delta_k=Y-T_train[i]
        temp=np.dot(W2.T,delta_k)
        derivative_val=derivative(A_new)
        delta_j=np.multiply(derivative_val,temp)
        #cnt+=1
 W1=W1-0.01*np.asarray(np.asmatrix(delta_j[1:]).T*np.asmatrix(train_x[i]))
 W2=W2-0.01*np.asarray(np.asmatrix(delta_k).T*np.asmatrix(A_new))

#ETerm1+=np.asarray(np.asmatrix(delta_j[1:]).T*np.asmatrix(train_x[i]))
        #ETerm2+=np.asarray(np.asmatrix(delta_k).T*np.asmatrix(A_new))
        #if cnt%10==0:
        #    W1=W1-0.01*ETerm1/10
        #    W2=W2-0.01*ETerm2/10
        #    ETerm1=np.zeros(W1.shape)
        #    ETerm2=np.zeros(W2.shape)
    E=0
    for k in range(10):
        E-=np.dot(T_train[i][k],np.log(Y[k]))
    print E
```

Uncomment for mini batch gradient descent.

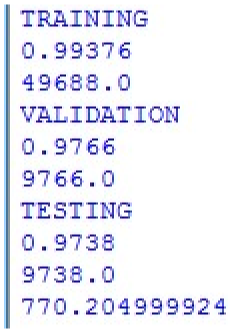
We need to set the initial weights to random in range [-1,1]. It is not advisable to set the initial weights to a constant number as it would fail.

```
W1=np.random.rand(100,785)
W1=W1*2*0.12
W1=W1-0.12
W2=np.random.rand(10,101)
W2=W2*2*0.12
W2=W2-0.12
```

Following is a graph obtained by plotting cross-entropy vs iterations.

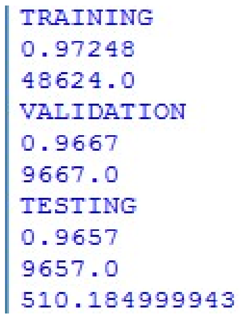


We obtained the following accuracy for 50 iterations:
Stochastic Gradient Descent:



```
TRAINING
0.99376
49688.0
VALIDATION
0.9766
9766.0
TESTING
0.9738
9738.0
770.204999924
```

The last line represents time taken.
Mini-batch stochastic gradient descent of batch size 10:

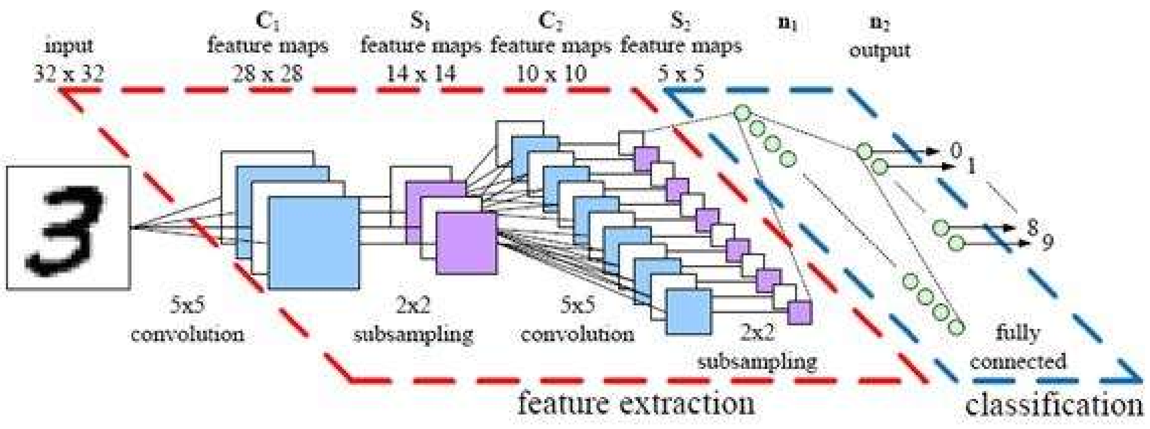


```
TRAINING
0.97248
48624.0
VALIDATION
0.9667
9667.0
TESTING
0.9657
9657.0
510.184999943
```

Convolution Neural Network

Convolution neural network is similar to Artificial Neural Network, except that it is majorly used in computer vision. Initially the image will be fed as input and the weights are kernels which are convolved with the image. After one step of convolution, the image is down sampled in size when it passes through a sub sampling layer (max-pooling layer). These two steps keep repeating till we arrive at 1x1 output where the output is treated as a scalar.

These scalar outputs will then be taken as inputs to the fully connected neural network with 1 or 2 hidden layers. This is the general architecture of Convolution Neural Networks.



Following is the training code to classify MNIST dataset using convolution neural networks

```
cross_entropy =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y_conv, y_))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
sess.run(tf.initialize_all_variables())
for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%100 == 0:
    train_accuracy = accuracy.eval(feed_dict={
        x:batch[0], y_: batch[1], keep_prob: 1.0})
    print("step %d, training accuracy %g"%(i, train_accuracy))
  train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
```

We obtain an accuracy of 99.2%

USPS dataset

The USPS dataset is a set of handwritten characters similar to the MNIST dataset. We use this dataset to prove the “No free lunch” theorem which states that a model trained on one dataset doesn’t necessarily work efficiently for other similar dataset.

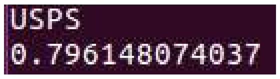
We load the USPS dataset using OpenCV. The steps involved are:

- Load the images in grey scale format
- Resize the image to 28x28
- Normalize the image by dividing the values by 255
- We then find the negative of the image by performing 1 – (image)
- Next we flatten the matrix to a vector of size 784

Once we have the vector of size 784 we can feed this to the trained model and check the accuracy obtained. I read the image using OpenCV and wrote the data obtained in form of a .h5 dataset. This can be read by installing h5py.

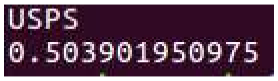
Following is the accuracy obtained for the MNIST trained models on USPS data

Logistic Regression



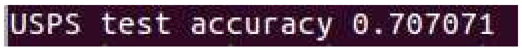
```
USPS
0.796148074037
```

Neural Network



```
USPS
0.503901950975
```

Convolution Neural Network

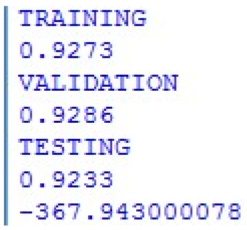


```
USPS test accuracy 0.707071
```

Results

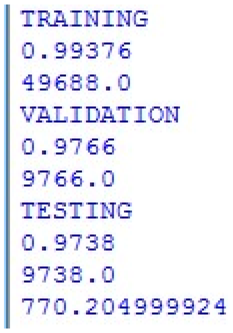
Following are the accuracy values for MNIST dataset

Logistic Regression: 50 iterations, min-batch Stochastic Gradient Descent of batch size of 10, learning rate of 0.01



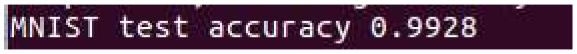
```
TRAINING
0.9273
VALIDATION
0.9286
TESTING
0.9233
-367.943000078
```

Neural Network: 50 iterations, Stochastic Gradient Descent, Learning rate 0.01



```
TRAINING
0.99376
49688.0
VALIDATION
0.9766
9766.0
TESTING
0.9738
9738.0
770.204999924
```

Convolution Neural Network: Batch size 100



```
MNIST test accuracy 0.9928
```