Recognizing Handwritten digits with Multilayer Perceptron

Xi Yu

Electrical and Computer Engineering Department
University of Florida
Gainesville, FL USA
yuxi@ufl.edu

***Abstract*—In this paper we recognize the handwritten digits based on the MNIST dataset with Multilayer Perceptron (MLP). We choose the architecture of the MLP and the number of the each layers in our neural networks and training the network's architecture in the training dataset. And we also improve the performance using different algorithms such as PCA, downsample for reducing the dimensionality, dropout and Stochastic Gradient Descent momentum for avoid overfitting and reducing the time of training. We show the performance of each algorithm in terms of learning curve and accuracy in the test dataset.**

Keywords—recognize the handwritten digits; MLP; reduce the dimensionality; Stochastic Gradient Descent (SGD);.

I. Introduction

The dataset which we use to train our multi-layer neural network is MNIST dataset. This dataset contains 60,000 handwritten digits images and each of them is 28*28 pixel. We use the first 50,000 handwritten digits images for training and the last 10,000 images for cross-validation to choose our hyper-parameter of our neural network such as the number of the units in hidden layer, the min-batch size, the iteration, and the learning rates. We also using other algorithms to reduce the dimensionality such as PCA and downsample the image in the dataset, using dropout and momentum to avoid overfitting and reduce the time of training, we calculate the accuracy and plot the learning curve and confusion matrix in another 10,000 images in test dataset to check the performance of our neural network.

This paper is organized as follows. In the section II, we introduce the mechanism of the MLP and our neural network architecture and how to deal with the MNIST dataset. In the section III, we present method of the reducing the dimensionality, for example PCA and downsample in our handwritten digits images in order to simplify our neural network architecture. The section IV is followed by presentation of the algorithm to improve the performance of the multi-layer neural network such as dropout and momentum. In the section V, we analyze influence of the parameters, such as learning rate, mini-batch size, the number of iteration, the number of the units in the hidden layer, the dimensionality of the images dataset, the dropout probability and the momentum parameter, and choose them in terms of the error and accuracy. We also compare the performance with different parameter. In the section VI, we drew a conclusion that how to choose the hyper-parameter of the MLP.

II. The Mechanism Of Multilayer Perceptron

A. The method to deal with the dataset.

The MNIST dataset contains 60,000 images and each of which is 28*28 pixel. We will regard each training input as a 784 dimensional vector. Each entry in the vector represents the gray value for a single pixel in the digits image. We use the sigmoid activation function, so we need normalize our input data, we transfer the gray value into 0 from 1, the formula we used is as follows.

$$X = \frac{X - X.\min}{X.\max - X.\min}$$

Here, X is a 60,000 by 784 matrix of the training dataset

Recognizing handwritten digits is a multi-label classification , we need to classify the dataset into 10 class. So the desired output is 10 dimension vector, and we use sigmoid as our activation. we need to deal with the label of our dataset. We can see in the Table1. For example, for a particular training images x, and the label of which is 7, then $y(x) = (0,0,0,0,0,0,0,1,0,0)^T$.

TABLE I. THE DESIRED OUTPUT

Digits label	**The output of the dataset**
0	$y(x) = (1,0,0,0,0,0,0,0,0,0,0)^T$
1	$y(x) = (0,1,0,0,0,0,0,0,0,0,0)^T$
2	$y(x) = (0,0,1,0,0,0,0,0,0,0,0)^T$
3	$y(x) = (0,0,0,1,0,0,0,0,0,0,0)^T$
4	$y(x) = (0,0,0,0,1,0,0,0,0,0,0)^T$
5	$y(x) = (0,0,0,0,0,1,0,0,0,0,0)^T$
6	$y(x) = (0,0,0,0,0,0,1,0,0,0,0)^T$
7	$y(x) = (0,0,0,0,0,0,0,1,0,0,0)^T$
8	$y(x) = (0,0,0,0,0,0,0,0,0,1,0)^T$
9	$y(x) = (0,0,0,0,0,0,0,0,0,0,1)^T$

B. The mechanism of the MLP

We train multilayer perceptrons for classification, we using one hidden layer because two or three hidden layers will too complex and there are many weights to train and cost too much time. The input of our data is 784 dimension so the number of the input layer is 784, and we classify the dataset into 10 class, so the number of the output layers is 10. we cannot determine the number of the hidden layer in this section, we will choose it in the section V. we can see the architecture in the Figure 1.

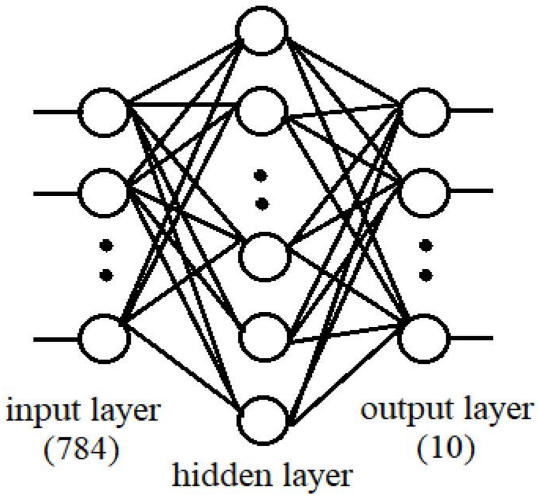


Figure 1 neural network architecture

We use backpropagation algorithm to update the weights between the perceptron elements. The backpropagation is common used in the gradient descent optimization algorithm to update the weights of the perceptron by computing the gradient of the cost function.

We suppose that we have only one hidden layers, and the cost function is

$$J = \frac{1}{2n} \sum_x (y(x) - output_2)^2$$

Where, y(x) is the desired output, and output2 represent the output of the last layer. And then we use the backpropagation algorithm to update the weights. In the beginning, we calculate the output in the forward direction, and then compute the gradient of the weights in the backward direction. The activation function is sigmoid. We add the bias into the weights. Here output1 represent the output of the hidden layer, net1 represent the input of the hidden layer, net2 represent the input of the output layer, and e represent the error between the output and desired.

$$net_1 = input \cdot w_1$$

$$output_1 = f(net_1)$$

$$net_2 = output_1 \cdot w_2$$

$$output_2 = f(net_2)$$

$$\frac{\partial J}{\partial w_2} = \frac{\partial J}{\partial e} \cdot \frac{\partial e}{\partial output_2} \cdot \frac{\partial output_2}{\partial w_2}$$

$$= -(y(x) - output_2) \cdot f'(net_2) \cdot output_2$$

$$= \delta_2 \cdot output_2$$

$$\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial output_1} \cdot \frac{\partial output_1}{\partial net_1} \cdot \frac{\partial net_1}{\partial w_1}$$

$$= \delta_2 \cdot w_2 \cdot f'(net_1) \cdot input$$

III. Methods Of Reducing Dimensionality

In this section, we introduce two way to lower the dimensionality of the MNIST dataset. One is PCA and the other is downsample. The advantages of small dimensionality is that we can train a little weights and get the optimal weight faster than that of high dimensionality, and we need less data to train our parameters.

A. PCA

PCA is a dimensionality reduction algorithm, in the problem of recognizing handwritten digits, PCA can transfer the 28*28 feature into a lower space. The main steps of PCA transform are as follows. First, we normalize the training data set and compute the covariance, means of them, and then find the eigenvectors of the covariance matrix, sort the eigenvectors by decreasing the eigenvalues. Finally, choose the K eigenvectors with largest eigenvalues, and return the component as the column of the matrix. We can see in the Figure 2, we reduce the dimensionality from 28*28 to 10*10.

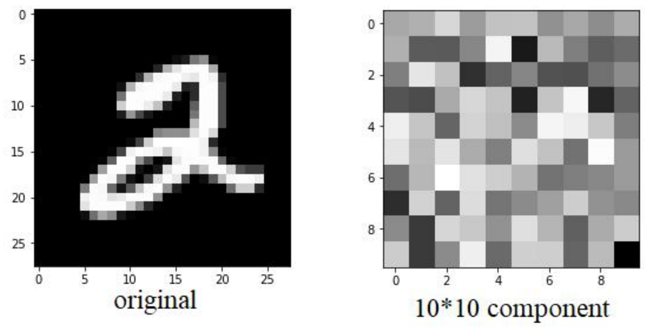


Figure 2 lower the dimensionality with PCA

B. downsample

There are many different downsample algorithms, and in this section, what we used is that downsample the image by taking average block wise. The main steps of this downsample algorithms are as follows. First, we choose a fixed factor, for example, we want to downsample a 4*4 pixel into 2*2 pixel, so the factor is 2, and then we can split the 4*4 pixel into 4 blocks, and each block contain 4 pixel. Finally, we average every block, and make up a new image. We can see in the Figure 3, we use this downsample method to transfer a the original digits into 14*14 pixel which we can see in the Figure 4.

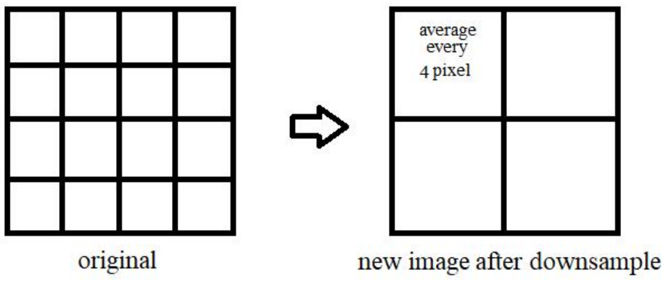


Figure 3 lower the dimensionality with downsample

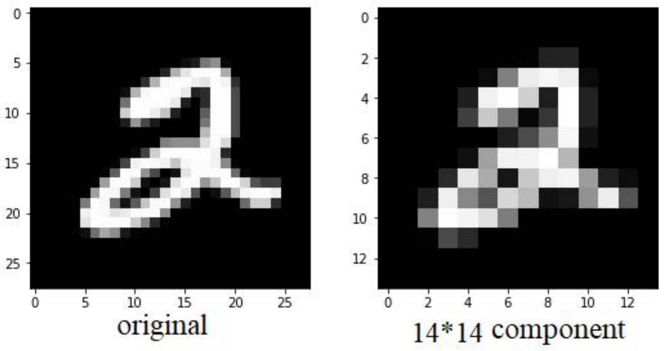


Figure 4 lower the dimensionality into 14*14

IV. Method Of Improving The Performance Of Mlp

In this section, we will introduce three way to improve the performance of Multilayer Perceptron (MLP). We know that a large and complex multilayer neural networks are more likely to overfitting and will slow to train those weights[1]. Besides, sometimes our neural network may in a local optimal and cannot find a global optimal. Therefore, we will use dropout to address the overfitting and use Stochastic Gradient Descent (SGD) to avoid local optimal problem, we can also add a momentum term to speed up the process of finding the optimal weights in our neural network.

A. dropout

The key idea of the dropout is to randomly drop out the perceptron elements in the process of training. There is no doubt that training a large and complex neural network is hard and requires a lot of computation and too much time to update weights, and more complex architecture make it possible to be overfitting. Moreover, a large amount of training dataset are needed to be trained in such complex architecture.

We know that model combination nearly always improve the performance of machine learning algorithm[1]. Therefore, dropout can deal with above problems, it prevents overfitting and removes the perceptron units of the input layer and hidden layers with a certain dropout probability in each iteration, so we training a different neural network architecture which is the combination of different neurons.

The main procedures to implement the dropout algorithm are as follows. First, we generate a set of random numbers (0 or 1) which obey the binomial distribution in a fixed probability, and then we multiply such random number set to the output of the hidden layers. We can see in the Figure 5. where p1, p2, p3 represent a number (0 or 1) [1].

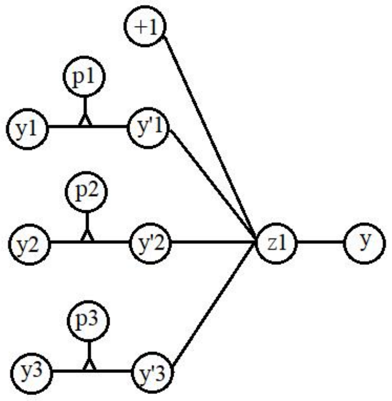


Figure 5 neural network with dropout

B. Stochastic Gradient Descent

In gradient descent we run all the samples of the training set to do a single update in each iteration, on the other hand in the stochastic gradient descent, every iteration we reorder the sample in the training dataset and select the subset of the training set to update our parameters, which also called min-batch stochastic gradient descent, it can avoid local optimal problem and run faster than update weights sample by sample. It also converge faster because we use only subset training sample and it starts improving itself right away from the first subset samples.

C. Momentum

Stochastic gradient descent with momentum allows neural network not only respond to the local gradient but also to the recent change trends in the error surface. Stochastic gradient descent with momentum depends on two parameters. One is the learning rate which is similar to the gradient descent, and the other is momentum parameter that defines the amount of momentum, which is set between 0 and 1. We can see in the follow formula.

$$w_{ji}(n) = w_{ji}(n) + \Delta w_{ji}(n)$$

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) + \alpha \Delta w_{ji}(n-1)$$

Here, η represent learning rate δ represent local gradient, α represent momentum parameter, Δw represent the change trends of the weights.

V. Choosing Hyper-parameter In MLP

In this section, we will choose the hyper-parameters of the Multilayer Perceptrons, for instance, the number of the units in hidden layer, the dimensionality of the downsample and PCA, the certain dropout probability, the learning rate, the min-batches size and momentum. We will also plot the learning curve and confusion matrix with different hyper-parameter, and compare each other with different algorithm.

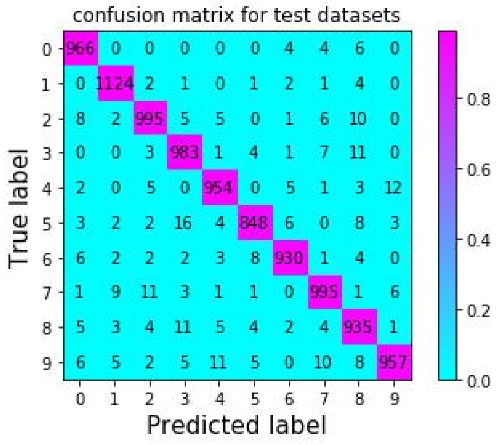
A. The number of units in hidden layer

We have already fixed the number of the input and output layer, the input layer contains 748 units and the output layer contains 10 units. And the activation function we used is sigmoid. We fixed the parameter of the learning rate, min-batch size and iteration. we choose the learning rate as 1 and min-batch size as 80, the iteration we select is 300. and then we change the number of the units in the hidden layer and compute the accuracy of them. We can see in the Table 2.

TABLE II. The Accuracy With Different Number Units

Number of units in hidden layer	**Accuracy (validation dataset)**	**Number of units in hidden layer**	**Accuracy (validation dataset)**
10	92.16%	60	95.42%
15	93.03%	65	95.7%
20	93.90%	70	87%
25	94.71%	75	95.76%
30	94.57%	80	96.01%
35	94.96%	85	95.24%
40	95.28%	90	95.77%
45	95.39%	95	95.9%
50	95.59%	100	97.01%
55	95.42%	105	96.7%

We can see in the Table 2, when the number of the units equal to 100, the accuracy in the validation dataset is highest than that of other number of units, and then we calculate the accuracy in the test dataset, and plot the confusion matrix in the Figure 6, and the accuracy in test dataset is 96.76%.



True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	966	0	0	0	0	0	4	4	6	0
1	0	1124	2	1	0	1	2	1	4	0
2	8	2	995	5	5	0	1	6	10	0
3	0	0	3	983	1	4	1	7	11	0
4	2	0	5	0	954	0	5	1	3	12
5	3	2	2	16	4	848	6	0	8	3
6	6	2	2	2	3	8	930	1	4	0
7	1	9	11	3	1	1	0	995	1	6
8	5	3	4	11	5	4	2	4	935	1
9	6	5	2	5	11	5	0	10	8	957

Figure 6 confusion matrix in test set

The Figure 6 show that the number in the diagonal is larger than that of in the off-diagonal, which means that the classification error is low, we can also find the classification error in different digital. The recognition rate of the digit "1" is maximal which is 99%, while the recognition rate of the digit "9" is the minimal, which is 94.8%. We also plot the learning curve and accuracy comparing training and validation dataset, which we can see in the Figure 7 and Figure 8.

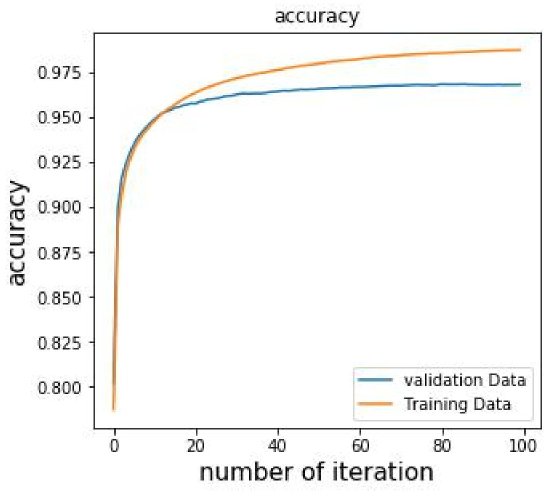


Figure 7 accuracy in the training and validation dataset

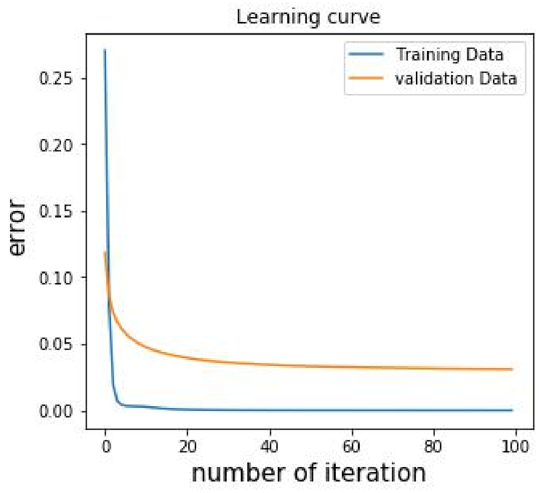


Figure 8 learning curve in training and validation dataset

B. Min-batches size and learning rate

In this section, we will choose the min-batches size and the learning rate in the fixed number of hidden layer and fixed iteration. Min-batches size and learning rate are two major parameters for Stochastic Gradient Descent. We calculate the accuracy in the validation data set in different pairs of min-batches size and learning rate, which we can see in Table 3.

TABLE III. THE ACCURACY IN TEST DATA SET

Accuracy (%)	Mini-batch size					
		10	**50**	**100**	**200**	**500**
Learning rate	**0.01**	92.9	87.51	83.33	63.57	49.96
	0.05	93.69	92.27	83.33	78.42	57.82
	0.1	94.56	93.12	92.56	91.58	90.55
	0.5	92.85	96.07	95.73	94.04	92.78
	1	92.16	95.68	97.01	93.43	88.76

The Table 3 shows that a large min-batches size allows a large learning rate. Because a larger min-batches reduce the variance of stochastic gradient updates, and in this condition we need to take a bigger leaning rate to converge quickly. We can also see in the Table that when min-batches size equal to 10, the best learning rate is 0.1 and when the learning rate equal to 1 the best mini-batch size is 100.

Therefore, when we choose a large min-batch size we should also select a large learning rate, and if we choose a small min-batch size, the learning rate which we select is also small.

C. The level of downsampling and the dimensionality of PCA

In this section, we will use downsample and PCA to reduce the dimensionality of the training data set and experiment with the different level of downsampling and dimensionality of the PCA. Another important thing is that when we use PCA in the training data set, we should also use the same eigenvectors which we compute in the training data set to project the test data set into the lower subspace, rather than compute the new eigenvectors in test data set separately.

TABLE IV. THE ACCURACY WITH PCA AND DOWNSAMPLE

Accuracy (test dataset)	**The size of the image after resize**			
	14*14	**7*7**	**4*4**	**2*2**
PCA	92.4%	93.76%	91.35%	64.83%
downsample	97.58%	96.29%	83.1%	47.67%

When we using downsample we can only resize the image into the factors of the 28, while when we using PCA, we can project the image into any lower dimensionality such as 10*10 and 8*8. Table 5 shows that the accuracy of PCA is lower than the downsample in the relative high dimensionality, while in the relative low dimensionality, the performance of the PCA is better than that of the downsample.

D. The dropout probability

In this section, we will use dropout in the hidden layer of the neural network, and we choose different dropout probability β and calculate the accuracy in the test data set. In the test dataset we should use the weights which multiply the dropout percent.We find that the accuracy in test data set is 95.58% when the dropout percent is 0.5, while the accuracy is 87.65% without dropout. We can see the learning curve in the training dataset in the Figure 9 and Figure 10, confusion matrix in the Figure 11. we choose the number of the hidden layer as 500, and learning rate is 1.5, the min-batch size we choose is 200, and we can see the accuracy is better when the dropout probability β is equal to 0.5 in the Table 5.

TABLE V. THE ACCURACY WITH DIFFERENT

	Dropout probability β			
	β = 0.1	**β = 0.3**	**β = 0.5**	**β = 0.8**
Accuracy (test dataset)	73.93%	86.64%	95.58%	93.35%

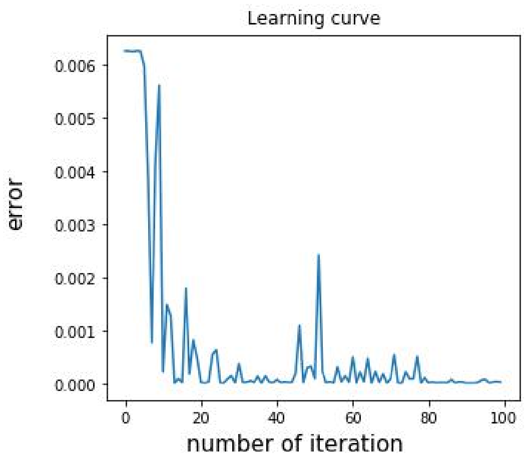


Figure 9 learning curve with dropout

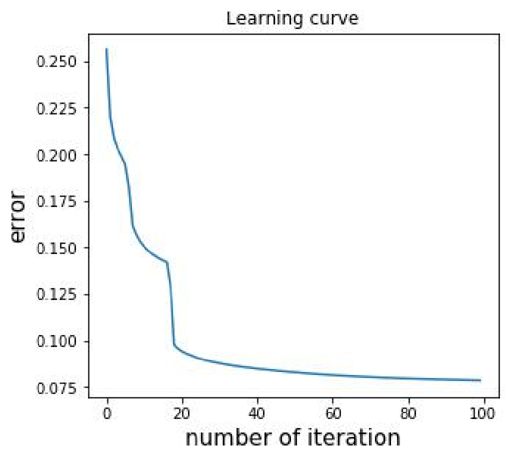


Figure 10 learning curve without dropout

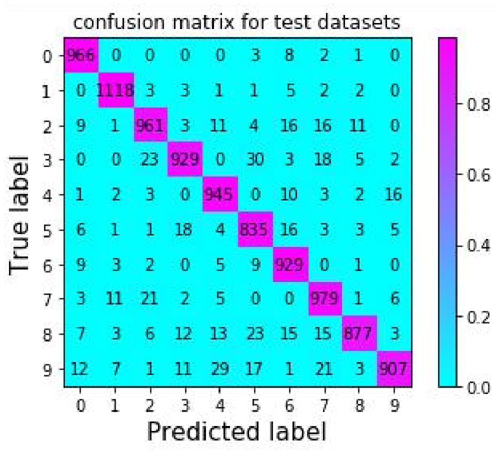


Figure 11 confusion matrix with dropout

E. Compare SGD and SGD-momentum

In this section, we will compare the performance with SGD and SGD-momentum. We will also experiment with different momentum parameter and plot the learning curve between the SGD and SGD-momentum. SGD oscillates across the slopes of the error surface and Momentum is a method that helps accelerate SGD in the relevant direction and reduce the oscillations[2]. We can see in the Figure 12. So we need less iteration with SGD-momentum, because it's faster toward to the optimal error.

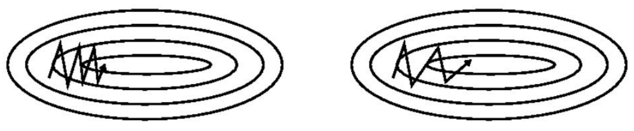


Figure 12 SGD and SGD with momentum

The Figure 13 shows that the error of the SGD-momentum is less than that of the SGD, because the SGD-momentum arrive at the optimal error faster and convergent quickly than SGD. The accuracy in the test data set is also better than that of the SGD. We can also see the accuracy in the test data set in the Figure 14.

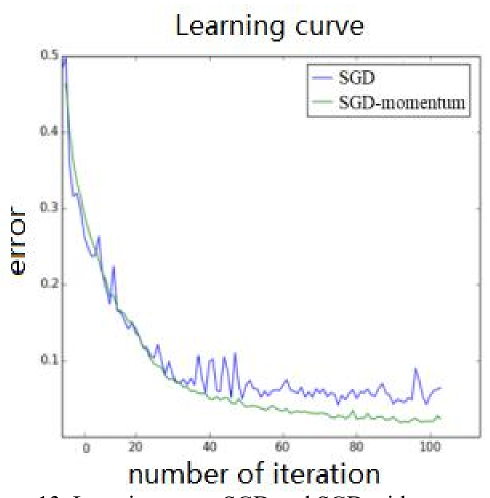


Figure 13 Learning curve SGD and SGD with momentum

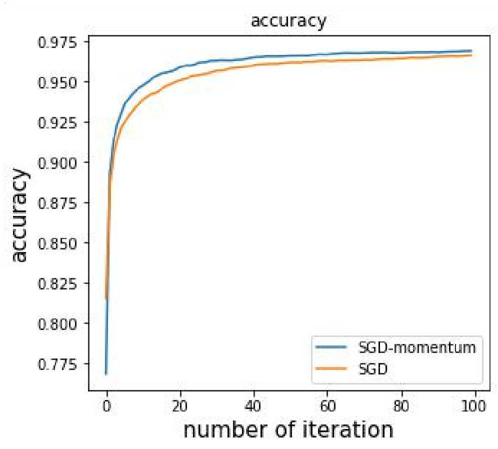


Figure 14 Accuracy between SGD and SGD with momentum

We fix learning rate, min-batch size and the number of neuron in the hidden layer, and then we experiment with the different momentum parameter α, in order to get the best accuracy in the test data set. We find that when α = 0.9, the accuracy is better in the test data set which we can see in the Table 6.

TABLE VI. THE ACCURACY WITH DIFFERENT MOMENTUM

Accuracy (test dataset)	**Momentum parameter α**			
	α = 0.1	**α = 0.3**	**α = 0.5**	**α = 0.9**
SGD-momentum	92.38%	94.15%	95.54%	96.58%

VI. CONCLUSION

This report is mainly about how to recognize the handwritten digital with multi-layer neural network with only one hidden layer, we choose the hyper-parameter of the neural network, and find the relationship between the hyper-parameter and the accuracy.

Firstly, we fix the learning rate, mini-batch size and the number of the input layer and select the number of the hidden layer to get a better accuracy. Secondly, we find the relationship between the min-batch size and learning rate, a large min-batch size needs to set a large learning rate, and a small min-batch size allows small learning rate, because a larger min-batch size weaken the variance of the gradient, therefore, we need a large step-size toward to the optimal error. And we also implement the dropout in the hidden layer, we find the accuracy is better when the dropout probability is 0.5.

We reduce the dimensionality of the training dataset with PCA and downsample, the performance of downsample is better than PCA while in relatively high dimensionality, while in lower dimensionality the PCA perform well. The accuracy is better when we use PCA to resize the image into the 7*7, and use downsample resize the image into 14*14.

Finally, we compare the performance between the SGD and SGD-momentum, we find that the SGD-momentum have a faster convergence and reduced oscillation in the learning curve. We also find that the accuracy will be better when momentum parameter equal to 0.9.

ACKNOWLEDGMENT

I confirm that this assignment is my own work, is not copied from any other person's work (published or unpublished), and has not been previously submitted for assessment either at University of Florida or elsewhere.

REFERENCES

[1] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky "Dropout: A Simple Way to Prevent Neural Network from Overfitting" Journal of Machine Learning Research 15(2014) 1929-1958

[2] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. Neural Networks : The Official Journal of the International Neural Network Society, 12(1), 145–151

[3] Sutton, R. S. (1986). Two problems with backpropagation and other steepest-descent learning procedures for networks. Proc. 8th Annual Conf. Cognitive Science Society

APPENDIX

```
import numpy as np
import os
import struct
import matplotlib
import matplotlib.pyplot as plt
import math
import textwrap
from array import array as pyarray
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.utils import shuffle
from sklearn.metrics import confusion_matrix

Dataset_path = '/Users/Xi Yu/Desktop/machine learning_homework/project2/'
def load_mnist(dataset="training", digits=np.arange(10), path= Dataset_path, size = 60000):
    if dataset == "training":
        fname_img = os.path.join(path, 'train-images.idx3-ubyte')
        fname_lbl = os.path.join(path, 'train-labels.idx1-ubyte')
    elif dataset == "testing":
        fname_img = os.path.join(path, 't10k-images.idx3-ubyte')
        fname_lbl = os.path.join(path, 't10k-labels.idx1-ubyte')
    else:
        raise ValueError("dataset must be 'testing' or 'training'")

    flbl = open(fname_lbl, 'rb')
    magic_nr, size = struct.unpack(">II", flbl.read(8))
    lbl = pyarray("b", flbl.read())
    flbl.close()

    fimg = open(fname_img, 'rb')
    magic_nr, size, rows, cols = struct.unpack(">IIII", fimg.read(16))
    img = pyarray("B", fimg.read())
    fimg.close()

    ind = [ k for k in range(size) if lbl[k] in digits ]
    N = size #int(len(ind) * size/100.)
    images = np.zeros((N, rows, cols), dtype=np.uint8)
    labels = np.zeros((N, 1), dtype=np.int8)
    for i in range(N): #int(len(ind) * size/100.)):
        images[i] = np.array(img[ ind[i]*rows*cols : (ind[i]+1)*rows*cols ])\
            .reshape((rows, cols))
        labels[i] = lbl[ind[i]]
    labels = [label[0] for label in labels]
    return images, labels, rows, cols

# Load training data
training_images, training_labels,rows, cols  = load_mnist('training')
testing_images,testing_labels,testing_rows,testing_cols  = load_mnist('testing')
train_label = np.array(training_labels)
test_label = np.array(testing_labels)

n_images = len(training_images)
n_labels = len(training_labels)
test_n_images = len(testing_images)
test_n_labels = len(testing_labels)

# To apply a classifier on this data, we need to flatten the image
# turn the data in a (samples, feature) matrix:
train_image = training_images.reshape((n_images, -1))
train_labels = train_label.reshape((n_labels, -1))

test_image = testing_images.reshape((test_n_images, -1))
test_labels = test_label.reshape((test_n_labels, -1))
features = train_image[:-1]
labels = train_labels[:-1]

#normalize the data
training_image = train_image/255
testing_image = test_image/255

Data = np.column_stack([training_image,train_labels])
# the number of the perceptron element of each layer
inputLayerSize = 784
hiddenLayerSize = 100
outputLayerSize = 10
mini_batchsize = 80
min_batch = np.int(50000/mini_batchsize)
eta = 1 # learning rate
iternation = 100
alpha = 0.9 # momentum

def ReLu(x):
        return np.maximum(x, 0)
def ReLuprime(x):
    return (x>0).astype(x.dtype)
#Sigmoid function
def sigmoid(x):
        return 1/(1+np.exp(-x))
```

```
#Gradient of sigmoid
def d_sig(x):
    return x*(1-x)

# the random initial value of weight and bias
W1 = np.random.uniform(-1,1,(inputLayerSize+1,hiddenLayerSize))
W2 = np.random.uniform(-1,1,(hiddenLayerSize+1,outputLayerSize))

def label(train_label, label_size):
    desired = np.zeros(10)
    for i in range(label_size):
        if train_labels[i]==0:
            desired_lable = np.array([1,0,0,0,0,0,0,0,0,0])
        if train_labels[i]==1:
            desired_lable = np.array([0,1,0,0,0,0,0,0,0,0])
        if train_labels[i]==2:
            desired_lable = np.array([0,0,1,0,0,0,0,0,0,0])
        if train_labels[i]==3:
            desired_lable = np.array([0,0,0,1,0,0,0,0,0,0])
        if train_labels[i]==4:
            desired_lable = np.array([0,0,0,0,1,0,0,0,0,0])
        if train_labels[i]==5:
            desired_lable = np.array([0,0,0,0,0,1,0,0,0,0])
        if train_labels[i]==6:
            desired_lable = np.array([0,0,0,0,0,0,1,0,0,0])
        if train_labels[i]==7:
            desired_lable = np.array([0,0,0,0,0,0,0,1,0,0])
        if train_labels[i]==8:
            desired_lable = np.array([0,0,0,0,0,0,0,0,1,0])
        if train_labels[i]==9:
            desired_lable = np.array([0,0,0,0,0,0,0,0,0,1])
        desired = np.row_stack([desired,desired_lable])
    Data_output = desired[1:label_size+1,:].T
    return Data_output

def forward(data,W1,W2):
    Input = np.zeros([10000,1])
    for i in range(10000):
        Input[i,0] = 1
    net1 = np.dot(data,W1)
    output1 = sigmoid(net1)
    #output1 = ReLu(net1)
    Output1 = np.column_stack([Input,output1])
    net2 = np.dot(Output1,W2)
    output2 = sigmoid(net2)
    return output2

#test in the validation data
def validation_accuracy():
    validation_data = train_image[50000:60000,:]
    validation_Data = validation_data /np.max(validation_data)
    validation = np.column_stack([validation_Input,validation_Data])
    DataOutput = train_label[50000:60000]
    output = forward(validation, W1,W2)
    predict = np.argmax(output, axis=1)
    count = 0
    for i in range(10000):
        if predict[i]==DataOutput[i]:
            count = count+1
    accuracy = count/10000
    return accuracy

Input = np.zeros([50000,1])
for i in range(50000):
    Input[i,0] = 1

validation_Input = np.zeros([10000,1])
for i in range(10000):
    validation_Input[i,0] = 1

mini_input = np.zeros([mini_batchsize,1])
for i in range(mini_batchsize):
    mini_input[i,0] = 1
pca = PCA(n_components=196, whiten=True)
data = pca.fit_transform(DataInput_training)
def plot_confusion_matrix(conf_arr,title,cmap=plt.cm.cool):
    norm_conf = []
    for i in conf_arr:
        a = 0
        tmp_arr = []
        a = sum(i, 0)
        for j in i:
            tmp_arr.append(float(j)/float(a))
        norm_conf.append(tmp_arr)

    fig = plt.figure()
    plt.clf()
    ax = fig.add_subplot(111)
    ax.set_aspect('equal')
    res = ax.imshow(np.array(norm_conf), cmap=cmap, interpolation='nearest')
    width, height = conf_arr.shape
    for x in range(width):
        for y in range(height):
            ax.annotate(str(conf_arr[x][y]),
            xy=(y, x),horizontalalignment='center',verticalalignment='center')
    fig.colorbar(res)
    alphabet = ['0','1','2','3','4','5','6','7','8','9']
    plt.xticks(range(width), alphabet[:width])
    plt.yticks(range(height), alphabet[:height])
    plt.ylabel('True label',fontsize=15)
    plt.xlabel('Predicted label',fontsize=15)
    plt.title(title)


#for i in range(30):
#np.random.shuffle(Data)
DataInput = Data[:,0:784]
DataOutput = Data[:,784]
DataInput_training = DataInput[0:50000,:]
Data_output = label(DataOutput,60000)
DataOutput_training = Data_output[:,0:50000].T
#b = DataInput_training/np.max(DataInput_training)
ArrayInput = np.column_stack([Input,DataInput_training])
#arrayInput = np.array(ArrayInput,dtype=np.int16)
J = np.zeros([mini_batchsize,mini_batchsize])
error = np.zeros(iternation)
```

```
ac = np.zeros(iternation)
dJdW1 = np.zeros([inputLayerSize+1,hiddenLayerSize])
dJdW2 = np.zeros([hiddenLayerSize+1,outputLayerSize])
for i in range(iternation):
    for j in range(min_batch):
        data = ArrayInput[mini_batchsize*j:mini_batchsize*(j+1),:]
        dataOutput_training
DataOutput_training[mini_batchsize*j:mini_batchsize*(j+1)]
        dJdW1Prev = dJdW1.copy()
        dJdW2Prev = dJdW2.copy()
        #forward
        net1 = np.dot(data,W1)
        output1 = sigmoid(net1)
        #output1 = ReLu(net1)
        Output1 = np.column_stack([mini_input,output1])
        net2 = np.dot(Output1,W2)
        output2 = sigmoid(net2)
        #backward
        error_output2 = dataOutput_training-output2
        de_output2 = np.array(d_sig(output2))
        delta3 = error_output2*de_output2
        dJdW2 = (Output1.T@delta3)/mini_batchsi
        delta2 = delta3@W2[1:hiddenLayerSize+1,:].T
        #de_output1 = np.array(ReLuprime(net1))
        de_output1 = np.array(d_sig(output1))
        local_error = de_output1*delta2
        dJdW1 = (data.T@local_error)/mini_batchsize
        #updata weights using momentum
        W2 = W2 + eta*dJdW2 + alpha*dJdW2Prev
        W1 = W1 + eta*dJdW1 + alpha*dJdW1Prev
    J = 0.5*error_output2@error_output2.T
    error_sum = 0
    for m in range(mini_batchsize):
        error_sum = J[m,m]
    error[i]= error_sum/mini_batchsize
    ac[i] = validation_accuracy()
    #print(ac)
    print(i)
```

```
n = np.arange(0,iternation,1)
fig = plt.figure(figsize=(5,10))
ax = fig.add_subplot(*[2,1,1])
p1 = ax.plot(n,ac)
#p2 = ax.plot(n,v_ac)
#plt.legend((p1[0],p2[0]),('SGD-momentum', 'SGD', ), fontsize=10)
#plt.ylabel('accuracy',fontsize=15)
#plt.xlabel('number of iteration', fontsize=15)
#plt.title('accuracy')

validation_data = training_image[50000:60000,:]
#validation_Data = validation_data /np.max(validation_data)
validation = np.column_stack([validation_Input,validation_data])
DataOutput = train_label[50000:60000]
output = forward(validation, W1,W2)
predict = np.argmax(output, axis=1)
count = 0
for i in range(10000):
    if predict[i]==DataOutput[i]:
        count = count+1
accuracy = count/10000
validation_d = np.zeros([10,10])
validation_d = confusion_matrix(DataOutput,predict)
plot_confusion_matrix
(d,'confusion matrix for training data sets',cmap=plt.cm.cool)

test_data = testing_image
test = np.column_stack([validation_Input,test_data])
test_DataOutput = test_label
test_output = forward(test, W1,W2)
test_predict = np.argmax(test_output, axis=1)
test_count = 0
for i in range(10000):
    if test_predict[i]==test_DataOutput[i]:
        test_count = test_count+1
test_accuracy = test_count/10000
test_d = np.zeros([10,10])
test _d = confusion_matrix(DataOutput,predict)
plot_confusion_matrix(d,'confusion matrix for test datasets',cmap=plt.cm.cool)
```