

پیاده سازی شبکه عصبی MLP با تابع فعال ساز sigmoid برای تشخیص handwritten digits:

```
import numpy as np
import random
import pandas as pd
import matplotlib.pyplot as plt
```

وارد کردن کتابخانه های لازم: numpy برای ضرب و جمع ماتریسی و برداری | random برای تولید وزن ها و بایاس های تصادفی | pandas برای خواندن mnist.csv و matplotlib برای رسم پیکسل های سیاه و سفید.

```
hidden_layers=[16,16]    #[2,3,2] three layers respectively with 2, 3, 2 neurons

weights=[]                #this is a list of a few matrices [weight matrices]
biases=[]                 #this is a list of vectors [biases]

activations=[-1]*(len(hidden_layers)+2)
                        #keeps track of activations at
                        #each layer (after activated by activation func)

num_of_inputs=784         # def = 784 (MNIST)
num_of_outputs=10         # def = 10 (MNIST)
alpha=0.01                #learning_rate
batch_size=500
```

ایجاد معماری خاص شبکه عصبی با 784 نورون در لایه ورودی 2 لایه مخفی هر کدام با 16 نورون و لایه ی خروجی با 10 نورون. همچنین در این مرحله پارامتر هایی مثل سایز بروزرسانی دسته ای - نرخ یادگیری تعیین می شوند. لیست Activations برای این هست که مقدار فعال شده ی نورون ها در هر لایه را در هر مرحله پیشخور ذخیره کند تا بتوانیم فرایند back propagation را انجام دهیم.

```

#generate biases for layer 2,3,...output
hidden_layers= hidden_layers + [num_of_outputs]

for i in range (len(hidden_layers)):
    temp=[]
    for k in range (hidden_layers[i]):|
        temp.append(random.uniform(-0.5,+0.5))
        #temp.append(random.uniform(0,1))#/2)
    biases.append(np.array(temp))

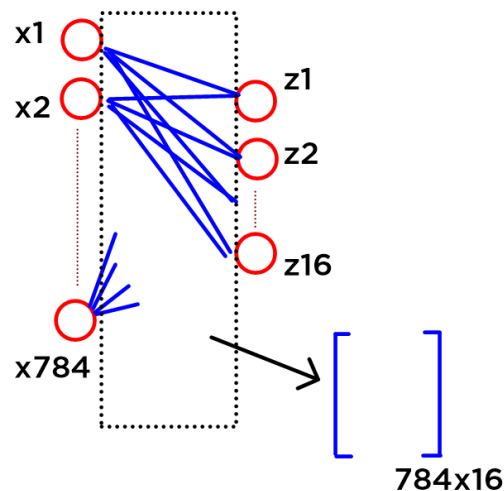
#10 output for MNIST dataset
#generate random weights inside the neural
hidden_layers=[num_of_inputs] + hidden_layers

for t in range(len(hidden_layers)-1):
    temp=[]
    for q in range (hidden_layers[t]*hidden_layers[t+1]):
        temp.append(random.uniform(-0.5,+0.5))
        #temp.append(random.uniform(0,1))#/2)

    matrix_temp=np.array(temp)
    matrix_temp=matrix_temp.reshape((hidden_layers[t],hidden_layers[t+1]))
    weights.append(matrix_temp)

```

در این قسمت ابتدا تمامی بایاس های تصادفی بین -0.5 و +0.5 ایجاد می کنیم و سپس ماتریس ها را می سازیم. این ماتریس ها در واقع همان وزن های روی یال های گراف هستند که با همان مقادیر بین -0.5 و +0.5 ساخته می شوند.



در واقع تا این مرحله weights لیستی از این ماتریس ها هست و biases لیستی از بردار های بایاس برای لایه های 2 به بعد هست.

```
#####
def sigmoid(x):                                # x=numpy so it affects all elements
    return 1/(1+np.exp(-x))

def feed_forward (input): #input np.array SHAPE=(1,784)
    activations[0]=input
    for t in range (len(weights)):
        next = np.dot(input,weights[t])
        next = next + biases[t]
        next = sigmoid (next)
        activations[t+1]=next
        input=next
    return input
```

تابع sigmoid تعریف شده و عمل پیشخور کردن با ضرب بردار ورودی از چپ به راست در ماتریس ها حساب می شود و خروجی یک بردار 10 تایی خواهد بود. نکته قابل توجه این هست که بعد از حساب کردن تابع sigmoid مقدار فعالسازی شده لایه ها در لیست activations ذخیره می شود.

قسمت بعدی برای back propagation هست.

```
def feed_backward (t_minus_y,alpha): #t_minus_y is a 1x10 numpy giving errors

    change_in_weights=[] #new List of matrices each element showing the delta_W
    change_in_biases=[] #new List of matrices each element showing the delta_B

    f_of_y_in=activations[-1]
    delta_k_all= t_minus_y * f_of_y_in * (1 - f_of_y_in)
    change_in_biases.append(alpha * delta_k_all)

    for k in range (2,len(hidden_layers)+1):
        z = activations[-k].reshape(1,-1)

        change_in_weights.append(alpha * np.dot(z.T,delta_k_all.reshape(1,-1)) )
        delta_k_in_all= np.dot(delta_k_all.reshape(1,-1),weights[-k+1].T)

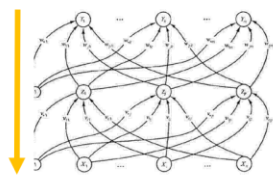
        f_of_z_in=activations[-k]
        delta_k_all= delta_k_in_all * f_of_z_in * (1- f_of_z_in)

        if k<len(hidden_layers):
            change_in_biases.append(alpha * delta_k_all)

    return change_in_weights, change_in_biases
```

این تابع مقدار خطا بین برچسب های واقعی و خروجی شبکه را می گیرد و با استفاده از مقدار نرخ یادگیری یک tuple بر می گرداند که عنصر اول آن یک لیست از ماتریس ها (تغییرات وزن ها) و عنصر دوم آن یک لیست از بردار ها (تغییرات بایاس ها) می باشد.

البته این تابع برای هر چند لایه ی دلخواه کار می کند و محدود فقط به 2 لایه ی مخفی نیست چون تو این تابع از for استفاده کرده ام که تا رسیدن به لایه ی ورودی تغییرات وزن ها و بایاس ها را حساب می کند. تمامی آپدیت ها با توجه به اسلاید ها نوشته شده اند:



پرسپترون چندلایه (الگوریتم آموزش)

□ پس انتشار خطا

- مرحله ۶- محاسبه خطا برای واحدهای خروجی (استفاده از الگوی هدف)

$$\delta_k = (t_k - y_k) f'(y_{in_k})$$

محاسبه پارامتر تصحیح وزن (بعداً در به روز کردن به کار می رود)

محاسبه پارامتر تصحیح بایاس (بعداً در به روز کردن به کار می رود)

ارسال δ_k (مقادیر دلتا) به واحدهای لایه قبلی (لایه مخفی)

- مرحله ۷- دریافت ورودی های دلتا توسط واحدهای مخفی از واحدهای خروجی

$$\delta_{in_j} = \sum_{k=1}^m \delta_k w_{jk}$$

ضرب در مشتق تابع فعال سازی جهت محاسبه پارامتر مربوط به اطلاعات خطا

محاسبه مقدار تصحیح وزن و بایاس (استفاده در به روز کردن)

$$\delta_j = \delta_{in_j} f'(z_{in_j})$$

$$\Delta w_{jk} = \alpha \delta_k z_j$$

$$\Delta w_{0k} = \alpha \delta_k$$

$$\Delta v_{ij} = \alpha \delta_j x_i$$

$$\Delta v_{0j} = \alpha \delta_j$$

پادگیری ژرف - دانشگاه صنعتی خواجه نصیرالدین طوسی

98

```
def weight_bias_updater (tuple): #(weights_changes, biases_changes)
    w=tuple[0]
    w.reverse()
    b=tuple[1]
    b.reverse()

    for t in range (len(b)):
        biases[t] = biases[t]+b[t]
        weights[t]= weights[t]+ w[t]
```

این تابع tuple خروجی از feed_backward را دریافت می کند و با توجه به آن لیست وزن ها و ماتریس ها را آپدیت می کند. (در واقع دلتا وزن ها و بایاس ها را به لیست وزن ها و بایاس های قبلی اضافه می کند.)

$$w_{jk}(new) = w_{jk}(old) + \Delta w_{jk}$$

$$v_{ij}(new) = v_{ij}(old) + \Delta v_{ij}$$

```
def One_Hot_Encoding(int):  
    if int==0:        return np.array([1,0,0,0,0,0,0,0,0,0])  
    elif int==1:      return np.array([0,1,0,0,0,0,0,0,0,0])  
    elif int==2:      return np.array([0,0,1,0,0,0,0,0,0,0])  
    elif int==3:      return np.array([0,0,0,1,0,0,0,0,0,0])  
    elif int==4:      return np.array([0,0,0,0,1,0,0,0,0,0])  
    elif int==5:      return np.array([0,0,0,0,0,1,0,0,0,0])  
    elif int==6:      return np.array([0,0,0,0,0,0,1,0,0,0])  
    elif int==7:      return np.array([0,0,0,0,0,0,0,1,0,0])  
    elif int==8:      return np.array([0,0,0,0,0,0,0,0,1,0])  
    elif int==9:      return np.array([0,0,0,0,0,0,0,0,0,1])  
  
def min_max_scale (array):  
    min=np.min(array)  
    max=np.max(array)  
    denom= max - min  
    if max==0:  
        return (np.array([0]*len(array)))  
    else:  
        return (array - min)/denom
```

تابع One_Hot_Encoding برچسب های داده های آموزش را به بردارهای 10 تایی تبدیل می کند تا قابل مقایسه با خروجی 10 تایی شبکه باشند. تابع min_max_scale رو هم زدیم که بتونیم ستون های دیتا ست رو scale کنیم. ولی در اجرا از آن استفاده ی نکردم.

```
def batch_adder (tup1, tup2):
    w1 = tup1[0]
    w2 = tup2[0]
    b1 = tup1 [1]
    b2 = tup2 [1]
    w3=[]
    b3=[]

    for i in range (len(w1)):
        w3.append(w1[i]+w2[i])
        b3.append(b1[i]+b2[i])

    return (w3,b3)

def batch_list_adder (list):
    my_list=copy.copy(list)
    for s in range (len(my_list)-1):
        my_list[s+1]=batch_adder(my_list[s],my_list[s+1])

    return my_list[-1]
```

این قسمت هم batch_adder دو تا tuple تغییرات دلتا را با هم جمع می کند و یک tuple تغییرات می دهد. همچنین تابع batch_list_adder هم یک لیستی از tuple های تغییرات دلتا را دریافت کرده و با کمک تابع batch_adder تمامی را جمع کرده و نهایتاً یک تغییرات کلی را بر می گرداند.

بسیار خوب حالا داده ها را می خوانیم. و سپس epoch ها را اجرا می کنیم.

من از داده های CSV استفاده کردم: [داده های آموزش](#) و [داده های آزمون](#)

```
df_train=pd.read_csv('mnist_train.csv')
#df_train=pd.read_csv('mnist_train.csv')[0:100]

df_train_labels= df_train['label'].values
df_train_labels2=np.array([]) #one hot encoded
for i in range (len(df_train_labels)):
    df_train_labels2 = np.append (df_train_labels2, One_Hot_Encoding(df_train_labels[i]))

#df_train_labels3 = df_train_labels2.reshape(60000,10)
df_train_labels3 = df_train_labels2.reshape(60000,10)

df_train_features = df_train.drop(columns=["label"])
df_train_features = df_train_features.values
```

در این قسمت با دستور های pandas داده ها را دیتا فریم قرار داده و سپس با دستور values. آن ها را به numpy (آماده برای فیت شدن روی مدل) در آورده ایم. همچنین برچسب ها را one_hot_encode کرده ایم تا آماده ی استفاده برای محاسبه ی مربعات خطا و فرایند back propagation باشند. حال epoch ها را اجرا می کنیم.

```

alpha=0.001
batch_size=500
for i in range (80): #epochs
    sum=0
    for t in range(0,60000,batch_size):
        list_of_change_tuples=[]
        #print('new_batch starting ... \n:')
        for b in range(batch_size):
            #print(t+b)
            input= df_train_features[t+b]
            #print(train_features[t], train_labels[t])
            output= feed_forward(input)
            error = df_train_labels3[t+b] - output
            change_tuple = feed_backward(error,alpha)
            list_of_change_tuples.append(change_tuple)
            #print(change_tuple)
            sum+=MSE(df_train_labels3[t+b],output)

        all_changes= batch_list_adder (list_of_change_tuples)
        weight_bias_updater(all_changes)

    print('epoch ', i, 'completed . . . squared error: ', sum)

```

در این قسمت تعداد epoch ها مشخص می شود و همچنین به تعداد batch_size تغییرات ذخیره و بعد از آن اعمال می شوند. مربعات خطا نیز در انتهای هر epoch حساب شده و گزارش می شود.

```

epoch 0 completed . . . squared error: 5552.278431456889
epoch 1 completed . . . squared error: 5324.41292721948
epoch 2 completed . . . squared error: 5247.634733357409
epoch 3 completed . . . squared error: 5104.506091480256
epoch 4 completed . . . squared error: 4887.057923427555
epoch 5 completed . . . squared error: 4671.330748090107
epoch 6 completed . . . squared error: 4457.294232860177
epoch 7 completed . . . squared error: 4242.43746848959
epoch 8 completed . . . squared error: 4028.441498802779
epoch 9 completed . . . squared error: 3808.2890051489367
epoch 10 completed . . . squared error: 3595.185160931191
epoch 11 completed . . . squared error: 3396.8857535946554
epoch 12 completed . . . squared error: 3218.2694737278925
epoch 13 completed . . . squared error: 3055.2990366515824
epoch 14 completed . . . squared error: 2879.091677983428
epoch 15 completed . . . squared error: 2731.1641339792286
epoch 16 completed . . . squared error: 2596.618086529119
epoch 17 completed . . . squared error: 2472.5390240708216
epoch 18 completed . . . squared error: 2380.324952663354
epoch 19 completed . . . squared error: 2274.3573493401955
epoch 20 completed . . . squared error: 2208.945933965802
epoch 21 completed . . . squared error: 2135.1589685989934

```



```

epoch 22 completed . . . squared error: 2066.761959086107
epoch 23 completed . . . squared error: 1955.8420458182816
epoch 24 completed . . . squared error: 1903.362303564675
epoch 25 completed . . . squared error: 1835.0372281968278
epoch 26 completed . . . squared error: 1787.9213446937276
epoch 27 completed . . . squared error: 1759.6967228941005
epoch 28 completed . . . squared error: 1713.4849121479801
epoch 29 completed . . . squared error: 1645.866938372004
epoch 30 completed . . . squared error: 1640.3025938765325
epoch 31 completed . . . squared error: 1617.0418409470863
epoch 32 completed . . . squared error: 1558.0151240424068
epoch 33 completed . . . squared error: 1516.9313396688506
epoch 34 completed . . . squared error: 1486.9432004514035
epoch 35 completed . . . squared error: 1471.1278935198131

epoch 36 completed . . . squared error: 1452.620108686149
epoch 37 completed . . . squared error: 1443.0984873747182
epoch 38 completed . . . squared error: 1397.1783711487788
epoch 39 completed . . . squared error: 1372.546259133494
epoch 40 completed . . . squared error: 1380.2957726939696
epoch 41 completed . . . squared error: 1365.3233897536973
epoch 42 completed . . . squared error: 1320.590530624907
epoch 43 completed . . . squared error: 1295.458639945018
epoch 44 completed . . . squared error: 1308.5653308254186
epoch 45 completed . . . squared error: 1284.2501659806494
epoch 46 completed . . . squared error: 1269.9021851255097
epoch 47 completed . . . squared error: 1292.4654171676686
epoch 48 completed . . . squared error: 1253.4088841206371
epoch 49 completed . . . squared error: 1247.3450773865027
epoch 50 completed . . . squared error: 1223.8594470474604
epoch 51 completed . . . squared error: 1220.3450094058383
epoch 52 completed . . . squared error: 1224.9611790304652
epoch 53 completed . . . squared error: 1173.3597807732726
epoch 54 completed . . . squared error: 1180.4162794833483
epoch 55 completed . . . squared error: 1181.755612670611
epoch 56 completed . . . squared error: 1192.2595125747087
epoch 57 completed . . . squared error: 1164.1197558811618
epoch 58 completed . . . squared error: 1157.5058215044864
epoch 59 completed . . . squared error: 1137.587652241558
epoch 60 completed . . . squared error: 1189.4669834256174
epoch 61 completed . . . squared error: 1154.1338854971316
epoch 62 completed . . . squared error: 1190.8123786833894
epoch 63 completed . . . squared error: 1143.544017094821
epoch 64 completed . . . squared error: 1132.3192020627082
epoch 65 completed . . . squared error: 1180.2854568095095
epoch 66 completed . . . squared error: 1185.048060696829
epoch 67 completed . . . squared error: 1141.8723085143765
epoch 68 completed . . . squared error: 1123.2420404207573
epoch 69 completed . . . squared error: 1077.9040272292953
epoch 70 completed . . . squared error: 1078.2496696902865
epoch 71 completed . . . squared error: 1085.0740327107999
epoch 72 completed . . . squared error: 1075.7187575488506
epoch 73 completed . . . squared error: 1073.2808124455225
epoch 74 completed . . . squared error: 1086.3997694466048
epoch 75 completed . . . squared error: 1075.8830004295255
epoch 76 completed . . . squared error: 1083.156375555399
epoch 77 completed . . . squared error: 1113.141815190442
epoch 78 completed . . . squared error: 1069.1314315081593
epoch 79 completed . . . squared error: 1093.2204589679113

```

حال درصد پیش بینی درست را مشخص می کنیم.


```
acc=0
for t in range (60000):
    if np.argmax(feed_forward(df_train_features[t]))== df_train_labels[t]:
        acc+=1
print("acc = ", acc/60000)
```

<ipython-input-125-06aa0043dc3d>:44: RuntimeWarning: overflow encountered in exp
return 1/(1+np.exp(-x))

acc = 0.8926833333333334

این دقت روی خود داده ی آموزش هست.

```
df_test=pd.read_csv('mnist_test.csv')
#df_train=pd.read_csv('mnist_train.csv')[0:100]

df_test_labels= df_test['label'].values
df_test_features = df_test.drop(columns=["label"])
df_test_features = df_test_features.values
```

```
acc=0
for t in range (10000):
    if np.argmax( feed_forward(df_test_features[t]) )== df_test_labels[t]:
        acc+=1
print("acc = ", acc/10000)
```

<ipython-input-125-06aa0043dc3d>:44: RuntimeWarning: overflow encountered in exp
return 1/(1+np.exp(-x))

acc = 0.8941

حال داده های تست را از mnist_test.csv می خوانیم و دقت را حساب کرده ایم.

روش معمولی

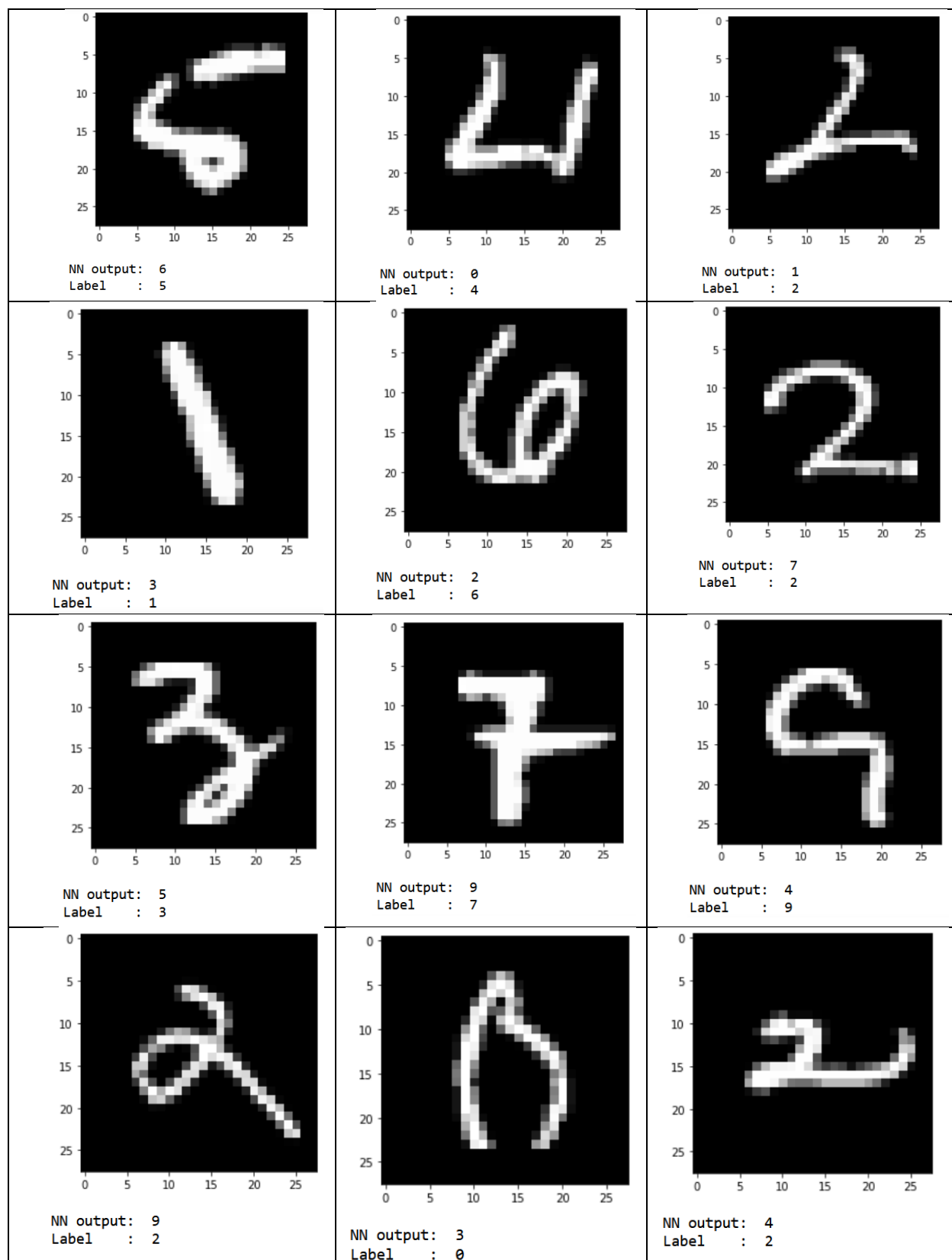
معماری شبکه عصبی	نرخ یادگیری	Batch size	تعداد epoch ها	مربع خطا	دقت در داده آموزش	دقت در داده آزمون
[784, 16, 16, 10]	0.001	500	80	1093	0.8926	0.8941
[784,200,80,10]	0.01	50	18	979	0.8988	0.9066

مورد دوم را هم مجزا بعداً حساب کردم.

کد زیر برای رسم چندتا از اشتباه های شبکه عصبی آموزش دیده هست:

```
for t in range (10000):
    if np.argmax( feed_forward(df_test_features[t]) )!= df_test_labels[t]:
        rl=df_test_labels[t]
        c=df_test_features[t]
        p=c.reshape((28,28))
        plt.imshow(p, cmap='gray')
        plt.show()
        print("NN output: ", np.argmax(feed_forward(df_test_features[t])))
        print("Label      : ", df_test_labels[t])
```

حال به چندتا از اشتباهات شبکه ی عصبی خود نگاه کنیم:



(تکنیک گشتاور) پیاده سازی شبکه عصبی MLP با تابع فعال ساز sigmoid برای تشخیص handwritten digits:

قسمتی از کد عوض می شود که توضیح می دهم:

```
alpha=0.001
batch_size=500
for i in range (45): #epochs
    sum=0
    for t in range(0,60000,batch_size):
        list_of_change_tuples=[]
        #print('new_batch starting ... \n:')
        for b in range(batch_size):
            #print(t+b)
            input= df_train_features[t+b]
            #print(train_features[t], train_labels[t])
            output= feed_forward(input)
            error = df_train_labels3[t+b] - output
            change_tuple = feed_backward(error,alpha)
            list_of_change_tuples.append(change_tuple)
            #print(change_tuple)
            sum+=MSE(df_train_labels3[t+b],output)

        all_changes= batch_list_adder (list_of_change_tuples)
        momentum_change = batch_adder_momentum (all_changes, keep_changes, mu=0.9)
        keep_changes = all_changes
        #weight_bias_updater(all_changes+ miu * keep_changes)
        weight_bias_updater(momentum_change)

    print('epoch ', i, 'completed . . . squared error: ', sum)
```

در این قسمت بعد از هر batch_size تغییرات قبلی ذخیره می شوند (keep_changes) و سپس تغییرات جدید با تغییرات قبلی که در میو μ ضرب شده جمع می شود و تابع updater با این تغییرات حاصل وزن ها و بایاس ها را بروزرسانی می کند.

در زیر تعریف توابع استفاده شده را می نویسیم:

```
# for momentum
k_weights=[]
k_biases=[]
#
```

```

#generate biases for layer 2,3,...output
hidden_layers= hidden_layers + [num_of_outputs]

for i in range (len(hidden_layers)):
    temp=[]
    temp2=[]
    for k in range (hidden_layers[i]):
        temp.append(random.uniform(-0.5,+0.5))
        temp2.append(0*random.uniform(-0.5,+0.5))
        #temp.append(random.uniform(0,1))#/2)
    biases.append(np.array(temp))
    k_biases.append(np.array(temp2))

#10 output for MNIST dataset
#generate random weights inside the neural
hidden_layers=[num_of_inputs] + hidden_layers

for t in range(len(hidden_layers)-1):
    temp=[]
    temp2=[]
    for q in range (hidden_layers[t]*hidden_layers[t+1]):
        temp.append(random.uniform(-0.5,+0.5))
        temp2.append(0*random.uniform(-0.5,+0.5))
        #temp.append(random.uniform(0,1))#/2)

    matrix_temp=np.array(temp)
    matrix_temp=matrix_temp.reshape((hidden_layers[t],hidden_layers[t+1]))
    weights.append(matrix_temp)
    k_weights.append(0*matrix_temp)

k_weights.reverse()
k_biases.reverse()
keep_changes = (k_weights, k_biases)

```

در این قسمت دو تا لیست `k_weights` و `k_biases` هم ساخته شده اند که همانطور که مشخص هست در ابتدا صفر هستند و از `epoch2` به بعد مقدار دهی می شوند تا تغییرات قبلی را ذخیره و ضریب ممان از آن را با تغییرات جدید جمع کنند. و این کار را تابع زیر انجام می دهد.

```

def batch_adder_momentum (tup1, tup2, mu=0.5):
    w1 = tup1[0]
    w2 = tup2[0]
    b1 = tup1 [1]
    b2 = tup2 [1]
    w3=[]
    b3=[]

    for i in range (len(w1)):
        w3.append(w1[i] + mu*w2[i])
        b3.append(b1[i] + mu*b2[i])

    return (w3,b3)

```

و نتیجه نهایی:

```

epoch 0 completed . . . squared error: 5497.363050685283
epoch 1 completed . . . squared error: 5181.676310864971
epoch 2 completed . . . squared error: 4802.319301780605
epoch 3 completed . . . squared error: 4373.1928797649125
epoch 4 completed . . . squared error: 3984.4473988120376
epoch 5 completed . . . squared error: 3597.04002723993
epoch 6 completed . . . squared error: 3223.6976961215
epoch 7 completed . . . squared error: 2913.999517098849
epoch 8 completed . . . squared error: 2654.755643855142
epoch 9 completed . . . squared error: 2443.886090524906
epoch 10 completed . . . squared error: 2314.2907249256036
epoch 11 completed . . . squared error: 2150.965894923678
epoch 12 completed . . . squared error: 2053.686266846971
epoch 13 completed . . . squared error: 1974.601748393316
epoch 14 completed . . . squared error: 1877.5090802617158
epoch 15 completed . . . squared error: 1773.7582077068998
epoch 16 completed . . . squared error: 1723.0392122792362
epoch 17 completed . . . squared error: 1733.71924227387
epoch 18 completed . . . squared error: 1686.3402704850314
epoch 19 completed . . . squared error: 1577.518575154226
epoch 20 completed . . . squared error: 1610.5508949203663
epoch 21 completed . . . squared error: 1517.2504632594453
epoch 22 completed . . . squared error: 1509.5311047106793
epoch 23 completed . . . squared error: 1423.2017283550538
epoch 24 completed . . . squared error: 1421.4896787809166
epoch 25 completed . . . squared error: 1396.0660244774758
epoch 26 completed . . . squared error: 1450.5182928859997
epoch 27 completed . . . squared error: 1440.5434436164824
epoch 28 completed . . . squared error: 1372.5029733252975
epoch 29 completed . . . squared error: 1377.5609654069372
epoch 30 completed . . . squared error: 1328.789858583588
epoch 31 completed . . . squared error: 1381.6741851104716
epoch 32 completed . . . squared error: 1290.9800908222776

```

روش ممان

معماری شبکه عصبی	نرخ یادگیری	Batch size	تعداد epoch ها	مربع خطا	دقت در داده آموزش	دقت در داده آزمون
[784, 16, 16, 10]	0.001	500	33	1290	0.8682	0.8681

همان طور که می بینیم روش گشتاور در تعداد epoch های کمتر به تقریباً همان درصد خوب بالای 85 روی داده ی آزمون و آموزش رسیده است. البته من تغییر خیلی زیادی رو نمی بینم شاید باید مقدار های batch_size بیشتر باشند.

(تکنیک دلتا_ بار_دلتا) پیاده سازی شبکه عصبی MLP با تابع فعال ساز sigmoid برای تشخیص handwritten digits:

برای این قسمت من نتونستم کدی بنویسم که درست کار کنه. همچنین از کتاب هم الگوریتم رو خوندم. صفحه 307 کتاب Laurene Fausett ولی چون مثالی هم نداشت خیلی متوجه نشدم چطور وزن ها متناسب با نرخ یادگیری متغیر آپدیت می شوند یعنی خود الگوریتم هم دقیقاً متوجه نشدم داره چی کار میکنه. بجای آن سعی کردم کمی با keras هم کار کنم ولی دلتا بار دلتا ظاهراً نبود و چند optimizer مختلف دیگر مثل stochastic gradient descent رو امتحان کردم.

+ Code + Text

```
[12] 1 from tensorflow.keras.models import Sequential
      2 from keras.layers import Dense, Flatten
      3 from tensorflow.keras.datasets import mnist

[13] 1 classifier = Sequential()

[14] 1 classifier.add(Flatten(input_shape=(28,28)))
      2 classifier.add(Dense(units=16,activation='sigmoid'))
      3 classifier.add(Dense(units=16,activation='sigmoid'))
      4 classifier.add(Dense(units=10,activation='sigmoid'))
      5 classifier.compile(optimizer='adam', loss='sparse_categorical_crossentropy',metrics=['accuracy'])
```

در این قسمت با استفاده از keras شبکه ی عصبی رو ساختیم. "Optimizer="adam" با روش stochastic گرادینان رو کاهش می دهد. تعداد لایه ها هم همان دو لایه مخفی 16 تایی و لایه ورودی 28x28=784 و لایه خروجی 10 تایی هست.

```
1 (X_train,y_train) , (X_test,y_test)=mnist.load_data()
```

از خود کتاب خانه ی mnist داده های آموزش و آزمون را لود می کنیم.

```
1 classifier.fit(X_train, y_train, batch_size=500, epochs = 50)
```

```
Epoch 1/50
120/120 [=====] - 1s 5ms/step - loss: 2.1022 - accuracy: 0.3527
Epoch 2/50
120/120 [=====] - 1s 4ms/step - loss: 1.7125 - accuracy: 0.6002
Epoch 3/50
120/120 [=====] - 1s 5ms/step - loss: 1.3973 - accuracy: 0.7028
Epoch 4/50
120/120 [=====] - 1s 5ms/step - loss: 1.1436 - accuracy: 0.7603
Epoch 5/50
120/120 [=====] - 1s 4ms/step - loss: 0.9504 - accuracy: 0.8180
Epoch 6/50
120/120 [=====] - 1s 4ms/step - loss: 0.8104 - accuracy: 0.8426
Epoch 7/50
120/120 [=====] - 1s 4ms/step - loss: 0.6992 - accuracy: 0.8563
Epoch 8/50
120/120 [=====] - 1s 4ms/step - loss: 0.6181 - accuracy: 0.8626
Epoch 9/50
120/120 [=====] - 1s 4ms/step - loss: 0.5619 - accuracy: 0.8691
Epoch 10/50
120/120 [=====] - 1s 4ms/step - loss: 0.5162 - accuracy: 0.8745
```

```
Epoch 11/50
120/120 [=====] - 1s 4ms/step - loss: 0.4817 - accuracy: 0.8789
Epoch 12/50
120/120 [=====] - 1s 4ms/step - loss: 0.4599 - accuracy: 0.8798
Epoch 13/50
120/120 [=====] - 1s 4ms/step - loss: 0.4491 - accuracy: 0.8783
Epoch 14/50
120/120 [=====] - 1s 4ms/step - loss: 0.4267 - accuracy: 0.8845
Epoch 15/50
120/120 [=====] - 1s 4ms/step - loss: 0.4170 - accuracy: 0.8847
Epoch 16/50
120/120 [=====] - 1s 4ms/step - loss: 0.4058 - accuracy: 0.8870
Epoch 17/50
120/120 [=====] - 1s 4ms/step - loss: 0.3946 - accuracy: 0.8884
Epoch 18/50
120/120 [=====] - 1s 4ms/step - loss: 0.3835 - accuracy: 0.8927
Epoch 19/50
120/120 [=====] - 1s 4ms/step - loss: 0.3758 - accuracy: 0.8930
Epoch 20/50
120/120 [=====] - 0s 4ms/step - loss: 0.3661 - accuracy: 0.8965
Epoch 21/50
120/120 [=====] - 1s 4ms/step - loss: 0.3675 - accuracy: 0.8943
Epoch 22/50
120/120 [=====] - 1s 5ms/step - loss: 0.3593 - accuracy: 0.8973
Epoch 23/50
120/120 [=====] - 1s 4ms/step - loss: 0.3492 - accuracy: 0.8993
Epoch 24/50
120/120 [=====] - 1s 5ms/step - loss: 0.3556 - accuracy: 0.8975
Epoch 25/50
120/120 [=====] - 1s 5ms/step - loss: 0.3481 - accuracy: 0.8982
Epoch 26/50
120/120 [=====] - 1s 5ms/step - loss: 0.3466 - accuracy: 0.8985
Epoch 27/50
120/120 [=====] - 1s 5ms/step - loss: 0.3434 - accuracy: 0.8998
Epoch 28/50
120/120 [=====] - 1s 5ms/step - loss: 0.3392 - accuracy: 0.9014
Epoch 29/50
120/120 [=====] - 1s 5ms/step - loss: 0.3349 - accuracy: 0.9015
Epoch 30/50
120/120 [=====] - 1s 5ms/step - loss: 0.3294 - accuracy: 0.9029
```



```
Epoch 31/50
120/120 [=====] - 1s 5ms/step - loss: 0.3248 - accuracy: 0.9053
Epoch 32/50
120/120 [=====] - 1s 5ms/step - loss: 0.3215 - accuracy: 0.9048
Epoch 33/50
120/120 [=====] - 1s 5ms/step - loss: 0.3298 - accuracy: 0.9039
Epoch 34/50
120/120 [=====] - 1s 5ms/step - loss: 0.3202 - accuracy: 0.9065
Epoch 35/50
120/120 [=====] - 1s 5ms/step - loss: 0.3105 - accuracy: 0.9097
Epoch 36/50
120/120 [=====] - 1s 5ms/step - loss: 0.3105 - accuracy: 0.9080
Epoch 37/50
120/120 [=====] - 1s 5ms/step - loss: 0.3097 - accuracy: 0.9072
Epoch 38/50
120/120 [=====] - 1s 4ms/step - loss: 0.3071 - accuracy: 0.9087
Epoch 39/50
120/120 [=====] - 1s 4ms/step - loss: 0.3134 - accuracy: 0.9064
Epoch 40/50
120/120 [=====] - 1s 4ms/step - loss: 0.3189 - accuracy: 0.9056
Epoch 41/50
120/120 [=====] - 1s 4ms/step - loss: 0.3001 - accuracy: 0.9115
Epoch 42/50
120/120 [=====] - 1s 4ms/step - loss: 0.2992 - accuracy: 0.9111
Epoch 43/50
Epoch 44/50
120/120 [=====] - 1s 4ms/step - loss: 0.2949 - accuracy: 0.9125
Epoch 45/50
120/120 [=====] - 1s 4ms/step - loss: 0.2941 - accuracy: 0.9127
Epoch 46/50
120/120 [=====] - 1s 4ms/step - loss: 0.3031 - accuracy: 0.9090
Epoch 47/50
120/120 [=====] - 1s 5ms/step - loss: 0.2924 - accuracy: 0.9133
Epoch 48/50
120/120 [=====] - 1s 4ms/step - loss: 0.2961 - accuracy: 0.9115
Epoch 49/50
120/120 [=====] - 1s 5ms/step - loss: 0.2968 - accuracy: 0.9110
Epoch 50/50
120/120 [=====] - 1s 5ms/step - loss: 0.2933 - accuracy: 0.9128
<keras.callbacks.History at 0x7fe55550abd0>
```

```
[25] 1 loss_on_test, acc_on_test = classifier.evaluate (X_test, y_test)
      2 print("out of sample acc is : ", acc_on_test)
```

```
313/313 [=====] - 1s 1ms/step - loss: 0.2973 - accuracy: 0.9129
out of sample acc is : 0.9128999710083008
```

روش stochastic gradient descent

دقت در داده آموزش	دقت در داده آزمون	sparse_categorical_crossentropy	تعداد epoch ها	Batch size	معماری شبکه عصبی
0.9128	0.9128	0.2933	50	500	[784, 16, 16, 10]