

تمرین عملی HW4

زهرا گنجی 9531802

سوال 1)

الف) پیاده سازی الگوریتم پرسپترون:

پیش پردازش داده ها:

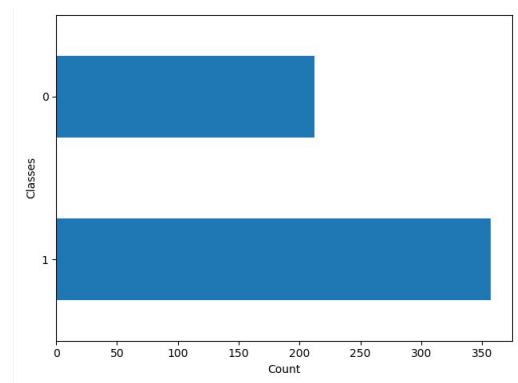
برای پیاده سازی این الگوریتم ابتدا از ماژول `sklearn.datasets` مجموعه داده `breast cancer` را لود می کنیم سپس متغیرهای ویژگی و هدف را با `breast_cancer.data` و `breast_cancer.target` میگیریم و در یک `pandas dataframe` قرار می دهیم.

```
#load the breast cancer data
breast_cancer = sklearn.datasets.load_breast_cancer()

#convert the data to pandas dataframe
data = pd.DataFrame(breast_cancer.data, columns = breast_cancer.feature_names)
data["class"] = breast_cancer.target
data.head()
data.describe()
```

این مجموعه داده شامل 569 مشاهده و 30 متغیر به جز متغیر هدف است. این مجموعه داده یک مجموعه ی نامتعادل است یعنی داده های در کلاس 0 خیلی کم تر از داده های کلاس 1 هستند و به صورت مساوی به نمایش در نمی آیند. این را می توانیم در نمودار زیر که فراوانی این دو کلاس را رسم کردیم ببینیم:

```
#plotting a graph to see class imbalance
data['class'].value_counts().plot(kind = "barh")
plt.xlabel("Count")
plt.ylabel("Classes")
plt.show()
```



در اینجا چون می خواهیم مدل پرسپترون ساده را پیاده سازی کنیم، از تکنیکی برای برقراری تعادل داده ها استفاده نمیکنیم. در واقع `efficient` نیست. از طرفی تفاوت تعداد داده های دو کلاس یا عدم تعادل داده ها در این دیتاست

خیلی زیاد نیست، تقریباً 200 داده در کلاس 0 و تقریباً 350 داده در کلاس 1 قرار دارند و در مواقعی که تفاوت اندک باشد از تکنیک های متعادل کردن داده ها استفاده نمی شود.

سپس داده ها را در محدوده ی $[0,1]$ scale می کنیم و سپس داده ها را به دو مجموعه test و train تقسیم می کنیم.

```
#perform scaling on the data.
X = data.drop("class", axis = 1)
Y = data["class"]
X = MinMaxScaler().fit_transform(X)
X = pd.DataFrame(X, columns=data.drop("class",axis = 1).columns)

#train test split.
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size = 0.1, stratify = Y, random_state = 1)
```

○ ساخت مدل:

در کلاس Perceptron در فایل Q1_part1_py ابتدا مقادیر w و b را در None constructor می دهیم، سپس چند کلاس داریم:

- این بخش به عنوان تابع استفاده نمی شود و در بخش های مورد نیاز در توابع دیگر به جای صدا زدن

```
for i in range(X.shape[0]):
    if np.dot(self.w, X.iloc[i]) >= self.b:
        result = 1
    else:
        result = 0
```

تابع خود الگوریتم نوشته می شود:

$X.iloc[i]$ هر نمونه ی X در مجموعه ی X است. هر نمونه در وکتور وزن ها ضرب نقطه ای می شود و اگر بیش تر از $threshold$ بود 1 در غیر اینصورت 0 بر می گرداند.

- تابعی که کل مجموعه ی X را میگیرد و به ازای هر نمونه ی X در این مجموعه Y را پیش بینی کرده و لیستی از پیش بینی ها Y را به عنوان خروجی می دهد.

```
# predictor to predict on the data based on w
def predict(self, X):
    Y = []
    # for x in X:
    # because model function is predicting y for each x
    for i in range(X.shape[0]):
        if np.dot(self.w, X.iloc[i]) >= self.b:
            result = 1
        else:
            result = 0
        Y.append(result)
    return np.array(Y)
```

- تابعی که بهترین وکتور وزن ها w و b را learn می کند. ورودی های این تابع :
input data(x,y), learning rate=0.3, number of epochs=10000 را به صورت ورودی می گیرد. اگر مقدار پیش بینی شده با مقدار اصلی متغیر هدف یکسان نباشد وزن ها را آپدیت می کند و این عمل را تکرار می کند تا الگوریتم همگرا شود یعنی وزن ها تغییر نکنند و داده ها درست کلاس بندی شوند.

در هر epoch میزان دقت اندازه گیری و نگه داشته می شود و در آخر بیش ترین دقت به عنوان max دقت مجموعه تست اعلام می گردد. و وزن و b بر اساس آن تعیین می شود.

```

def fit(self, X, Y, epochs, lr):
    self.w = np.ones(X.shape[1])
    self.b = 0
    accuracy = {}
    max_accuracy = 0
    wt_matrix = []
    # for all epochs
    for i in range(epochs):
        for count in range(X.shape[0]):
            if np.dot(self.w, X.iloc[count]) >= self.b:
                y_pred = 1
            else:
                y_pred = 0
            # compare real class value and prediction and update weights till congestion
            if Y.iloc[count] == 1 and y_pred == 0:
                self.w = self.w + lr * X.iloc[count]
                self.b = self.b - lr * 1
            elif Y.iloc[count] == 0 and y_pred == 1:
                self.w = self.w - lr * X.iloc[count]
                self.b = self.b + lr * 1

        wt_matrix.append(self.w)
        accuracy[i] = accuracy_score(self.predict(X), Y)
        if (accuracy[i] > max_accuracy):
            max_accuracy = accuracy[i]

```

```

wt_matrix.append(self.w)
accuracy[i] = accuracy_score(self.predict(X), Y)
if (accuracy[i] > max_accuracy):
    max_accuracy = accuracy[i]
    chkptw = self.w
    chkptb = self.b

# checkpoint (Save the weights and b value)
self.w = chkptw
self.b = chkptb
print(max_accuracy)
return np.array(wt_matrix)

```

○ اجرا و ارزیابی:

مدل را با داده های train می سازیم سپس کارایی مدل را با دقت مجموعه ی test اندازه می گیریم:

```

# Building Model
perceptron = Perceptron()

#epochs = 10000 and lr = 0.3
print("max accuracy in train")
wt_matrix = perceptron.fit(X_train, Y_train, 10000, 0.3)

#making predictions on test data
Y_pred_test = perceptron.predict(X_test)

#checking the accuracy of the model
print("test accuracy")
print(accuracy_score(Y_pred_test, Y_test))

```

```

test size: 0.100000
max accuracy in train:
0.982421875
test accuracy:
0.9473684210526315

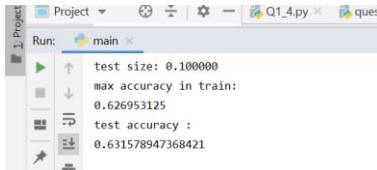
```

○ کارایی بیش تر:

در این مرحله پارامترهای مختلف را امتحان می کنیم تا بهترین و بهینه ترین پارامترها که منجر به دقت های بالاتری می شوند را پیدا کنیم.

- به جای استفاده از تابع `fit` برای پیدا کردن بهترین W ، اگر W را به صورت رندوم مقدار دهی کنیم در `test size`های مختلف داریم: (این قسمت در کد کامنت شده است)

```
# predictor to predict on the data based on w
def predict(self, X):
    Y = []
    # Take random weights in your model and test the result.
    self.w = np.random.randint(low=10, high=100, size=X.shape[1])
    # because model function is predicting y for each x
    for i in range(X.shape[0]):
        if np.dot(self.w, X.iloc[i]) >= self.b:
            result = 1
        else:
            result = 0
        Y.append(result)
    return np.array(Y)
```



که در مقایسه با دقت اصلی مدل بسیار کم تر است. پس با بهترین w که الگوریتم `learn` می کند دقت بالاتری داریم.

- سائز مجموعه ی `test` را تغییر می دهیم و بهترین سائز را (که در آن دقت بالاتر باشد) را انتخاب می کنیم، برای مثال اگر الگوریتم را در 4 سائز مختلف 0.1، 0.2، 0.3، 0.4 ران کنیم داریم: (این قسمت در کد کامنت است)

```
15 # Further optimization
16 # Vary the train-test size split and see if accuracy changes.
17 for i in range(1,5):
18     test_size = 1/10
19     print()
20     print("test size: %f"%test_size)
21 #train_test_split.
22 X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=test_size, stratify=Y, random_state=1)
23
24 # Building Model
25 perceptron = Perceptron()
26 wt_matrix = perceptron.fit(X_train, Y_train, 10, 0.3)
27
28 # making predictions on test data
29 Y_pred_test = perceptron.predict(X_test)
30
31 # checking the accuracy of the model
32 print("test accuracy :")
33 print(accuracy_score(Y_pred_test, Y_test))
34
35
```

```

test size: 0.100000
max accuracy in train:
0.982421875
test accuracy:
0.9473684210526315

test size: 0.200000
max accuracy in train:
0.9824175824175824
test accuracy:
0.9736842105263158

test size: 0.300000
max accuracy in train:
0.9824120603015075
test accuracy:
0.9473684210526315

test size: 0.400000
max accuracy in train:
0.9853372434017595
test accuracy:
0.9605263157894737

Process finished with exit code 0

```

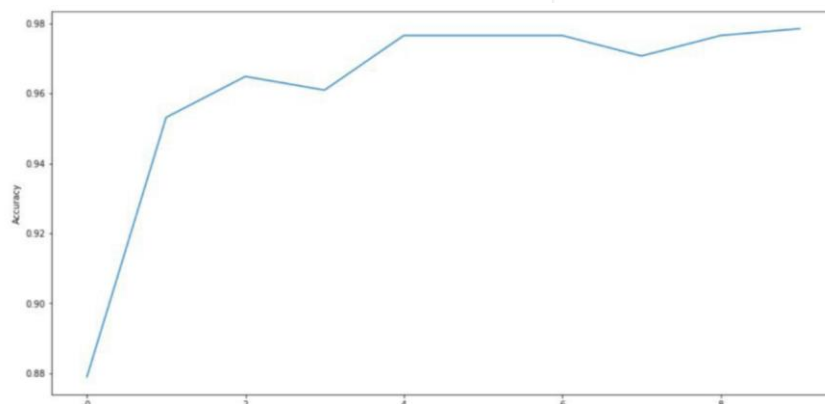
در test size=0.2 بیش ترین دقت را داریم.

- نمایش دادن دقت در epochsهای مختلف در نمودار:

```

lists= sorted(accuracy.items())
a,b = zip(*lists)
# plot the accuracy values over epochs
plt.plot(a,b)
plt.xlabel("Epoch #")
plt.ylabel("Accuracy")
# plt.ylim([0, 1])
plt.show(block=False)

```



در نهایت با افزایش epoch دقت افزایش می یابد.

- نمایش دقت در learning rateهای مختلف (0.1 و 0.2 و 0.3) (این قسمت در کد کامنت است)

```

# Choose Larger "learning rates".test on the model and visualize the change in accuracy.
acc=[]
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.1, stratify=Y, random_state=
# Building Model
perceptron = Perceptron()
for i in range(1, 4):
    print("lr= %f" % (i / 10))
    wt_matrix = perceptron.fit(X_train, Y_train, 10, i / 10)

    # making predictions on test data
    Y_pred_test = perceptron.predict(X_test)

    # checking the accuracy of the model
    acc[i]=accuracy_score(Y_pred_test, Y_test)
    print("test accuracy :")
    print(accuracy_score(Y_pred_test, Y_test))

lists= sorted(acc.items())
a,b = zip(*lists)
# plot the accuracy values over epochs
plt.figure(figsize=(16, 8))
plt.plot(a,b)
plt.xlabel("learning rates")
plt.ylabel("Accuracy")
plt.ylim([0, 1])

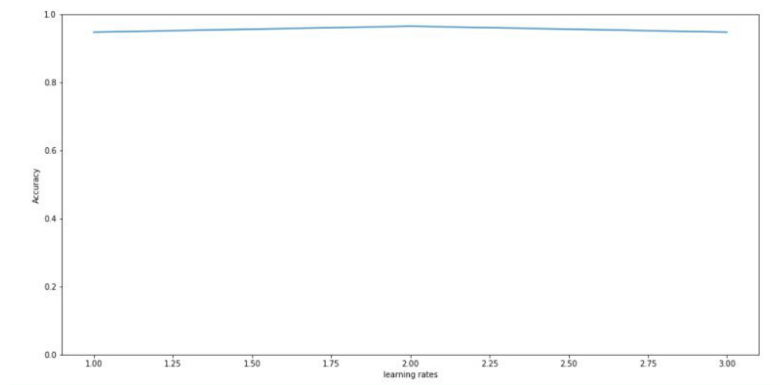
```

```

C:\Users\yasus\pycharm\projects\yaws_data\mining\venv\scripts
max accuracy in train:
0.982421875
test accuracy :
0.9473684210526315
lr= 0.100000
max accuracy in train:
0.978515625
test accuracy :
0.9473684210526315
lr= 0.200000
max accuracy in train:
0.978515625
test accuracy :
0.9649122807017544
lr= 0.300000
max accuracy in train:
0.982421875
test accuracy :
0.9473684210526315
Process finished with exit code 0

```

نمودار:



(ب)

حال می خواهیم یک شبکه عصبی با یک لایه ی پنهان را طراحی کنیم کارکرد آن روی جداسازی داده ها را با پرسپترون مقایسه کنیم.

در این قسمت مقادیر bias را 0 در نظر گرفته ام. اگر 0 نبود در هر مرحله در fit ضرب نقطه ای w در x با b جمع می شد و b نیز در تابع update fit می شد :

```
b.assign_sub(delta_b*r)
```

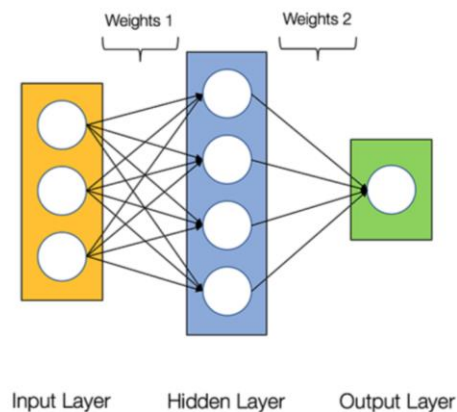
و دقت مدل زمانی که b ، fit شود و بهترین مقدار انتخاب شود بالاتر می رود.

برای پیش پردازش داده ها که شامل load کردن داده ها، تبدیل به pandas dataframe و scale کردن آن ها و جداسازی به مجموعه های test و train مانند قسمت قبل عمل می کنیم.

ساخت مدل شبکه عصبی:

کلاس NuralNetwork:

شبکه عصبی با یک لایه پنهان:



همان طور که مشاهده می شود دو سری وزن داریم. در constructor این کلاس یک ورودی به نام layer داریم که یک لیست با 3 المان است : layer[input_dim,units,output_dim] در اینجا تعداد units را برابر input_dim قرار داده ایم.

تولید وزن های اولیه:

در constructor 2 سری وزن به صورت رندوم تولید کرده و در لیست self.weights قرار می دهیم.

```
# Set weights
self.weights = []
for i in range(1, len(layers) - 1):
    r = 2 * np.random.random((layers[i - 1] + 1, layers[i] + 1)) - 1
    self.weights.append(r)

    r = 2 * np.random.random((layers[i] + 1, layers[i + 1])) - 1
    self.weights.append(r)
```

تابع fit:

این تابع با گرفتن مجموعه های X_{train} و Y_{train} و $epochs=10000$ و $learning_rate=0.2$ در ورودی، بهترین w را به داده ها fit می کند و آن را در $self.w$ قرار می دهد تا مدل برای پیش بینی داده های جدید از این w استفاده کند. هدف اصلی یافتن بهترین w و b است که تابع $loss$ به کم ترین مقدار خور برسد.

Fastforward:

در این تابع در هر دور (تعداد تکرار ها به تعداد $epochs$) با ضرب نقطه ای هر نمونه ورودی در w و اعمال تابع $activation$ روی نتیجه که در نوع آن را در $constructor$ تعیین کردیم، مقدار پیش بینی شده برای آن نمونه را بدست می آورد. سپس مقدار $loss$ یا همان $error$ ها را محاسبه می کنیم.

The output \hat{y} of a simple 2-layer Neural Network is:

$$\hat{y} = \sigma(W_2 \sigma(W_1 x + b_1) + b_2)$$



```
for k in range(epochs):
    i = np.random.randint(X.shape[0])
    a = [X[i]]

    for l in range(len(self.weights)):
        dot_value = np.dot(a[l], self.weights[l])
        activation = self.activation(dot_value)
        a.append(activation)

    # output Layer
    error = y.iloc[i] - a[-1]
    deltas = [error * self.activation_prime(a[-1])]

    # we need to begin at the second to last Layer
    # (a layer before the output Layer)
    for l in range(len(a) - 2, 0, -1):
        deltas.append(deltas[-1].dot(self.weights[l].T) * self.activation_prime(a[l]))

    # reverse
    # [Level3(output)->Level2(hidden)] => [Level2(hidden)->Level3(output)]
    deltas.reverse()
```

Backpropagation:

بعد از اندازه گیری $error$ پیش بینی خود، دنبال راهی برای $propagate$ کردن $error$ ها هستیم برای آپدیت کردن وزن ها.

برای دانستن مقدار مناسب تغییر در وزن ها برای آپدیت کردنشان از مشتقات تابع $loss$ بر حسب w استفاده می کنیم. اگر این مشتق ها را داشته باشیم وزن ها را با آن افزایش و کاهش می دهیم و به این کار گرادیان نزولی می گوییم. گرچه ما

به طور مستقیم نمی توانیم مشتق loss را بر حسب همه ی وزن ها بدست آوریم و از قانون زنجیره ای استفاده می

$$Loss(y, \hat{y}) = \sum_{i=1}^n (y - \hat{y})^2$$

$$\frac{\partial Loss(y, \hat{y})}{\partial W} = \frac{\partial Loss(y, \hat{y})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial W} \quad \text{where } z = Wx + b$$

$$= 2(y - \hat{y}) * \text{derivative of sigmoid function} * x$$

$$= 2(y - \hat{y}) * z(1-z) * x$$

کنیم.

در کد داریم:

```
...
# backpropagation
# 1. Multiply its output delta and input activation
# to get the gradient of the weight.
# 2. Subtract a ratio (percentage) of the gradient from the weight.
for i in range(len(self.weights)):
    layer = np.atleast_2d(a[i])
    delta = np.atleast_2d(deltas[i])
    self.weights[i] += learning_rate * layer.T.dot(delta)
```

تابع predict : از این تابع برای پیش بینی خروجی داده های مجموعه ی test با وزن های fit شده استفاده می گردد.

اگر مقدار پیش بینی شده بیش تر از 0.5 باشد آن نمونه در کلاس 1 و در غیر این صورت آن نمونه در کلاس 0 قرار دارد.

```
def predict(self, x):
    # print(x)
    a = np.concatenate((np.ones(1).T, np.array(x)),axis=None)
    for l in range(0, len(self.weights)):
        a = self.activation(np.dot(a, self.weights[l]))
    if a > 0.5:
        p=1
    else:
        p=0
    return p
```

سپس دقت مدل را برای مجموعه داده های تست به دست می آوریم.

```
units = X_train.shape[1]
nn = NeuralNetwork([X_train.shape[1],1])
nn.fit(X_train,Y_train)
prediction=[]

for i in range(X_test.shape[0]):
    prediction.append(nn.predict(X_test.iloc[i]))

for i in prediction:
    print(i)

# checking the accuracy of the model
print("test accuracy")
print(accuracy_score(prediction, Y_test))
```

نتیجه:

```

1
1
test accuracy
0.631578947368421

Process finished with exit code 0

```

در قسمت اول (با perceptron) اگر self.b را 0 در نظر بگیریم داریم:

```

HW4_Q1_main_part1.py 15 def predict(self, X):
HW4_Q1_part1.py 16     self.b=0
HW4_Q1_part2.py 17     Y = []
HW4_Q2_part1.py 18     # Take random weights in your model and test the result.
HW4_Q2_part2.py 19     Perceptron -> predict()

Run: main x
C:\Users\Asus\PycharmProjects\HW3_datamining\venv\Scripts\python.exe C:/Users/Asus/PycharmProjects/HW3_datamining/HW4_Q1_main_part1.py
max accuracy in train:
0.6328125
test accuracy :
0.5263157894736842

Process finished with exit code 0

```

پس مشاهده می شود در کل دقت مدل nural network با یک لایه پنهان از perceptron بیش تر است. در perceptron تابع تصمیم گیری یک step function است و خروجی باینری است و فقط برای داده هایی که به صورت خطی جدا پذیر باشند مناسب است. در nural network از activation function ها استفاده می شود و این باعث ایجاد عدد حقیقی در خروجی می شود که معمولا بین 0 و 1 یا 1- و 1 هستند. مدل nural network با لایه پنهان از تعدادی لایه و نورون به همراه توابع activation استفاده می کند و قدرتمند تر از perceptron قادر به تشخیص داده هایی است که از نظر خطی جداپذیر نیستند یا به وسیله ی hyper plane جداپذیر هستند. علاوه بر این MLP Network ها انعطاف پذیری بالاتری نسبت به perceptron دارند. پس دقت بالاتری نسبت به perceptron دارند. همچنین perceptron به دلیل داشتن افزونگی کارایی کم تر و سرعت کم تری دارد.

سوال (2)

```

10
11 df1=pd.read_csv("../question2/Dataset1.csv")
12 df2=pd.read_csv("../question2/Dataset2.csv")

```

خوشه بندی با استفاده از الگوریتم kmeans:

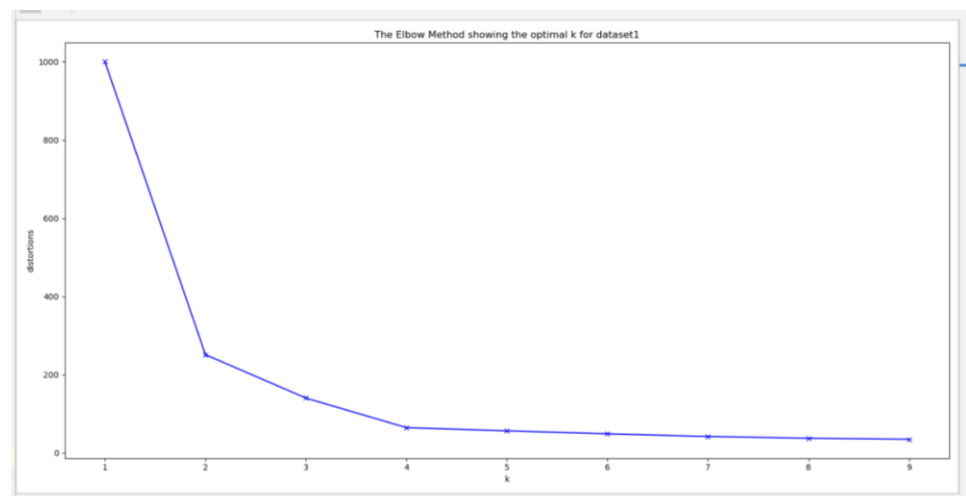
روش Elbow:

ابتدا با روش elbow تعداد k مناسب را بدست می آوریم. بر اساس این روش یک نمودار میکشیم که محور عمودی (y-axis) واریانس (مجموع توان دوم فاصله هر نقطه از مرکز کلاستر خودش) و محور افقی (x-axis) تعداد k است و بر اساس این نمودار مشاهده می شود که هر چه k بیش تر می شود distortion یا همان واریانس یا cost کم تر میشود

چون خوشه ها منسجم تر و بهتر می شوند. جایی که شیب نمودار به صورت خیلی مشهود تغییر کند k را مشخص می کند.

روش elbow برای dataset1:

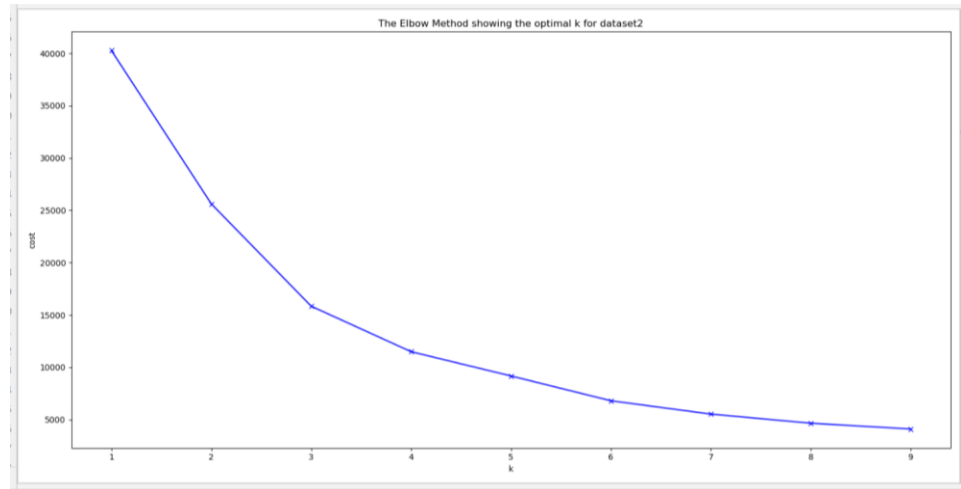
```
13
14 # elbow method for dataset1
15 distortions = []
16 K = range(1,10)
17 for k in K:
18     kmeanModel = KMeans(n_clusters=k)
19     kmeanModel.fit(df1)
20     distortions.append(kmeanModel.inertia_)
21 plt.figure(figsize=(16,8))
22 plt.plot(K, distortions, 'bx-')
23 plt.xlabel('k')
24 plt.ylabel('distortions')
25 plt.title('The Elbow Method showing the optimal k for dataset1')
26 plt.show()
```



مشاهده می کنیم در $k=2$ تغییر شیب شدید است .

روش elbow برای dataset2:

```
# elbow method for dataset2
distortions = []
K = range(1,10)
for k in K:
    kmeanModel = KMeans(n_clusters=k)
    kmeanModel.fit(df2)
    distortions.append(kmeanModel.inertia_)
plt.figure(figsize=(16,8))
plt.plot(K, distortions, 'bx-')
plt.xlabel('k')
plt.ylabel('cost')
plt.title('The Elbow Method showing the optimal k for dataset2')
plt.show(block=False)
```



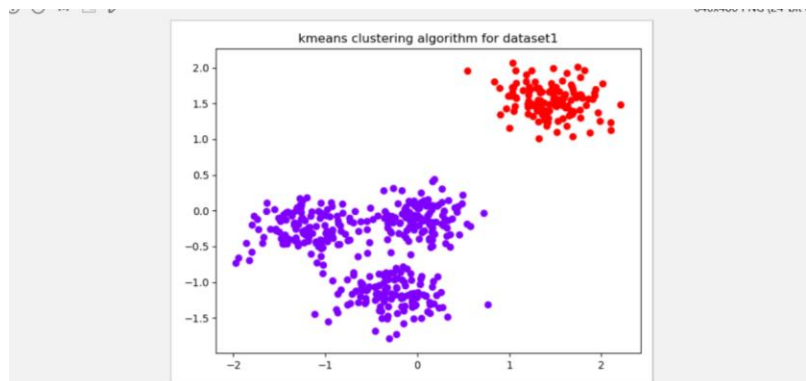
مشاهده می کنیم در $k=4$ تغییر شیب شدید است .

در این نمودار تغییر شدید در نمودار خیلی واضح نیست و این باعث می شود که احتمال خطا در تشخیص k بیش تر باشد و تعیین k برای $kmeans$ دچار مشکل شود. پس این می تواند یکی از دلایل خوب نبودن $kmeans$ برای این دیتاست باشد.

$kmeans$ برای dataset1:

$K=2$

```
# Kmeans for dataset1
k=2
kmeans = KMeans(n_clusters=k, random_state=0).fit(df1)
print(kmeans.labels_)
print(kmeans.cluster_centers_)
plt.scatter(df1['X'], df1['Y'], c=kmeans.labels_, cmap='rainbow')
plt.title('kmeans clustering algorithm for dataset1')
plt.show(block=False)
```



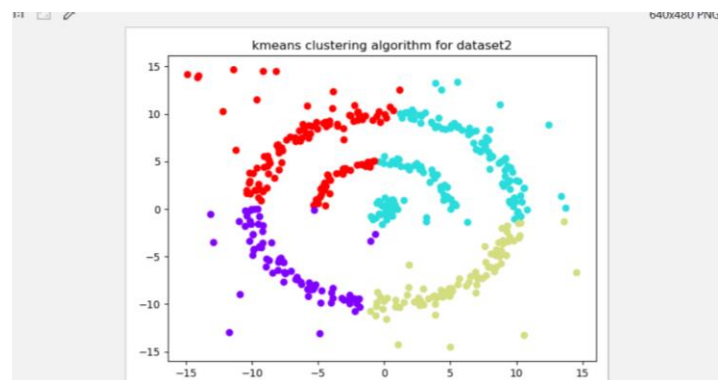
یک ضعف $kmeans$ در این دیتاست مشخص کردن k در ابتدا است و به همان تعداد خوشه ایجاد می شود. چون $k=2$ است دقیقاً دو خوشه ایجاد می شود در صورتی که می شود بهترین k اتوماتیک تعیین شود.

علاوه بر این kmeans چون میانگین داده ها را حساب می کند به داده های نویز و پرت حساس است و مقادیر خیلی زیاد و کم مراکز و میانگین در نتیجه خوشه را تحت تاثیر قرار می دهند و قادر به تشخیص داده های پرت نیست و آن ها را هم جزو خوشه ها قرار داده است.

Kmeans برای dataset2:

K=4

```
# Kmeans for dataset2
k=4
kmeans = KMeans(n_clusters=k, random_state=0).fit(df2)
print(kmeans.labels_)
print(kmeans.cluster_centers_)
plt.scatter(df2['X'],df2['Y'], c=kmeans.labels_, cmap='rainbow')
plt.title('kmeans clustering algorithm for dataset2')
plt.show(block=False)
```



همان طور که مشاهده می شود الگوریتم kmeans در خوشه بندی این دیتاست مناسب نیست زیرا kmeans برای تشخیص خوشه های با اشکال غیر محدب مناسب نیست پس نمی تواند خوشه های با شکل های مختلف را شناسایی کند چون بر اساس میانگین مراکز را میابد و خوشه بندی را انجام می دهد و یک الگوریتم partitioning است. الگوریتم های بر مبنای چگالی مانند dbscan این مشکل را رفع می کنند.

علاوه بر این kmeans چون میانگین داده ها را حساب می کند به داده های نویز و پرت حساس است و مقادیر خیلی زیاد و کم مراکز و میانگین در نتیجه خوشه را تحت تاثیر قرار می دهند و قادر به تشخیص داده های پرت نیست. ضعف دیگر لزوم مشخص کردن k در ابتدا است و به همان تعداد خوشه ایجاد می شود.

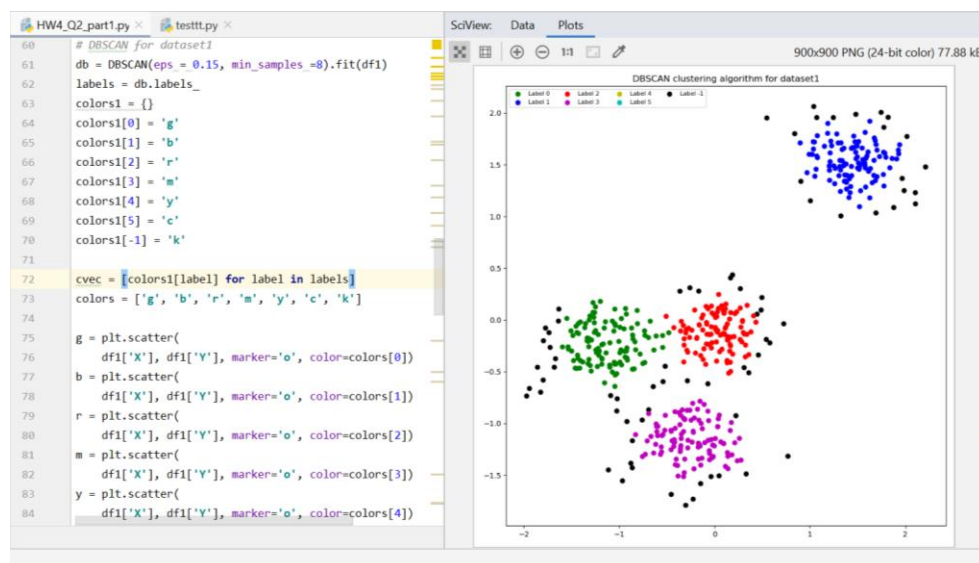
الگوریتم خوشه بندی DBSCAN:

در این الگوریتم با تعداد minpts (min_samples) مساوی اگر eps را افزایش دهیم یا اگر با eps مساوی minpts را کاهش دهیم، تعداد خوشه ها بیش تر میشود و داده های پرت بهتر تشخیص داده می شود و خوشه های با اشکال بهتری ایجاد می شود. در غیر اینصورت تعداد خوشه ها بیش تر می شود و داده های پرت نیز درون خوشه ها قرار می گیرند.

DBSCAN برای dataset1:

Eps:0.15

Min_samples=8

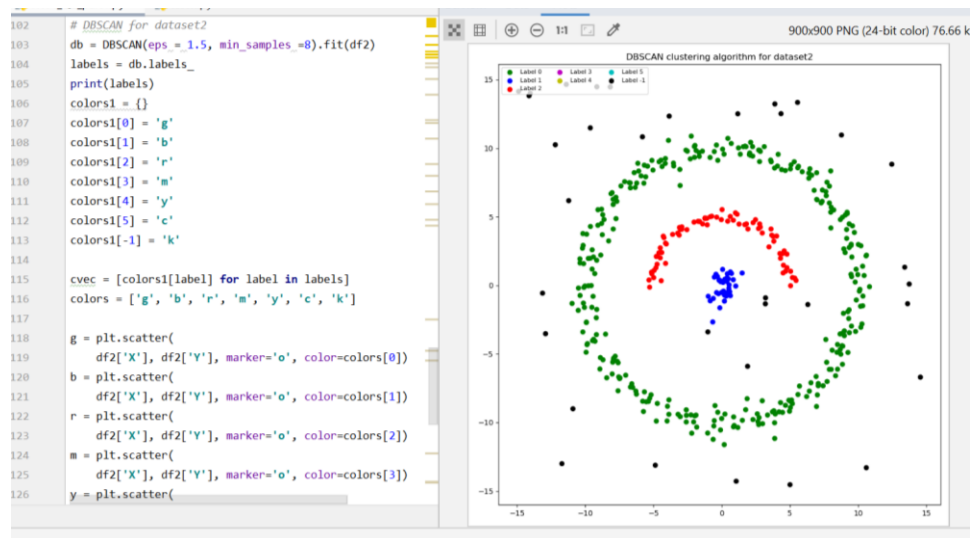


DBSCAN در این دیتاست ضعف kmeans را برطرف کرده و داده های نویز را تشخیص داده (نقاط مشکی) و اتوماتیک بر اساس eps و minpts بهترین k انتخاب شد. پس خوشه ها بهتر و منسجم تر هستند.

DBSCAN برای dataset2:

Eps=1.5

Min_samples=8



DBSCAN در این دیتاست ضعف kmeans را برطرف کرده یعنی قادر به تشخیص خوشه های با اشکال مختلف و غیر محدب است و داده های نویز تأثیری در خوشه بندی نگذاشته اند و نویز ها را تشخیص داده است (نقاط مشکلی). علاوه بر آن بهترین k در این الگوریتم اتوماتیک بر اساس eps و minpts انتخاب شد.

(2)

رسم dendrogram:

```
def plot_dendrogram(model, **kwargs):
    # create the counts of samples under each node
    counts = np.zeros(model.children_.shape[0])
    n_samples = len(model.labels_)
    for i, merge in enumerate(model.children_):
        current_count = 0
        for child_idx in merge:
            if child_idx < n_samples:
                current_count += 1 # leaf node
            else:
                current_count += counts[child_idx - n_samples]
        counts[i] = current_count

    linkage_matrix = np.column_stack([model.children_, model.distances_,
                                      counts]).astype(float)

    # Plot the corresponding dendrogram
    dendrogram(linkage_matrix, **kwargs)
```

از روش سلسله مراتبی agglomerative برای خوشه بندی استفاده می کنیم و در ابتدا threshold را 0 می گذاریم تا به صورت کامل پیش برود و سپس یک threshold تعریف می کنیم و درخت را از آن جا میبریم تا خوشه ها بدست آیند.

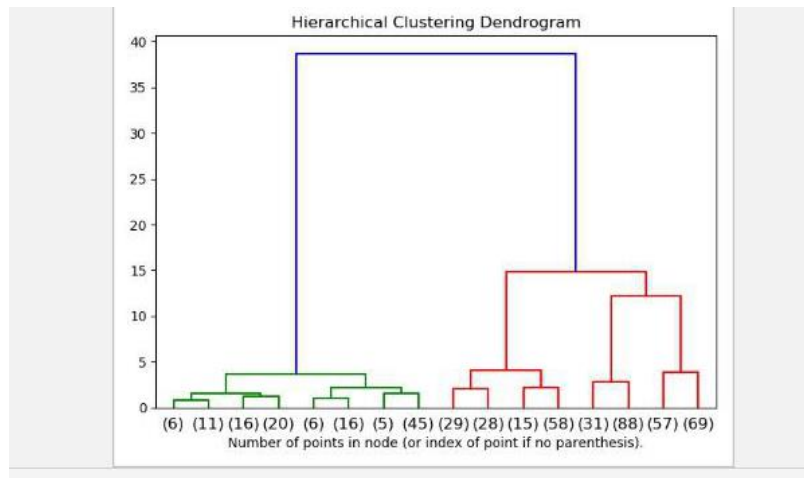
در اینجا 3 سطح بالای درخت را در نظر گرفتیم و از آن جا درخت را بریدیم.

```

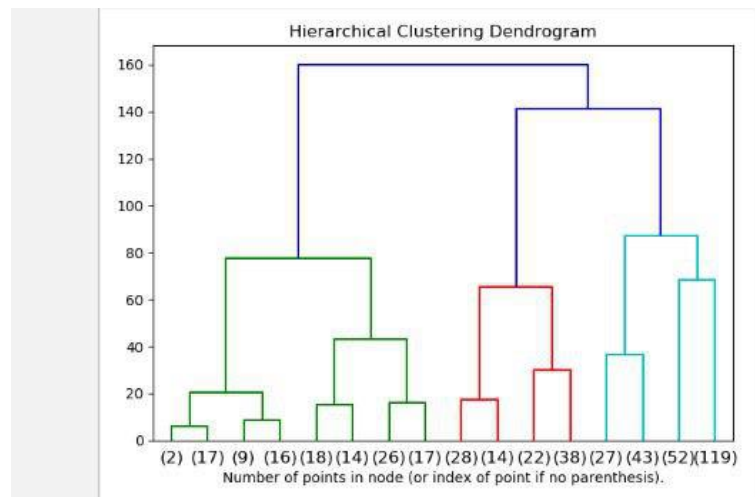
27
28 X=pd.read_csv('./question2/Dataset1.csv')
29 # setting distance_threshold=0 ensures we compute the full tree.
30 model = AgglomerativeClustering(distance_threshold=0, n_clusters=None).fit(X)
31 plt.title('Hierarchical Clustering Dendrogram for dataset1')
32 # plot the top three levels of the dendrogram
33 plot_dendrogram(model, truncate_mode='level', p=3)
34 plt.xlabel("Number of points in node (or index of point if no parenthesis).")
35 plt.show(block=False)
36
37 X=pd.read_csv('./question2/Dataset2.csv')
38 # setting distance_threshold=0 ensures we compute the full tree.
39 model = AgglomerativeClustering(distance_threshold=0, n_clusters=None).fit(X)
40 plt.title('Hierarchical Clustering Dendrogram for dataset2')
41 # plot the top three levels of the dendrogram
42 plot_dendrogram(model, truncate_mode='level', p=3)
43 plt.xlabel("Number of points in node (or index of point if no parenthesis).")
44 plt.show(block=False)

```

(dataset1 برای Dendrogram



(dataset2 برای Dendrogram



از مزیت های روش سلسله مراتبی این است که نیازی نیست در ابتدا تعداد خوشه ها یعنی k را به عنوان ورودی به الگوریتم بدهیم. ولی یک شرط خاتمه نیاز دارد و پیدا کردن $threshold$ مناسب برای برش نمودار دشوار است.

از معایب دیگر آن می توان به این اشاره کرد در این کاری را که قبلا کردیم یعنی خوشه بندی های قبلی را نمی توان undo کرد و این که مقیاس پذیری خوبی ندارد، پیچیدگی زمانی اش حداقل $O(n^2)$ است (n تعداد کل object ها)

سوال 3)

در این سوال می خواهیم داده های MNIST را با شبکه عصبی دسته بندی کنیم.

ابتدا داده ها را لود می کنیم:

```
# the data, shuffled and split between train and test sets
data=tf.keras.datasets.mnist.load_data()
(X_train, Y_train), (X_test, y_test) = data

# create a validation set
```

قسمت اول) در این قسمت بعد از load داده ها از `tf.keras.datasets.mnist.load_data()` ، از مجموعه ی train مجموعه ی validation را می سازیم.

```
# create a validation set
X_valid, X_train = X_train[:5000] / 255.0, X_train[5000:] / 255.0
y_valid, y_train = Y_train[:5000], Y_train[5000:]
```

در این قسمت داده ها را با شبکه ی fully connected یا همان dense در keras دسته بندی می کنیم.

```
# Create a model using Keras API
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(50, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

برای train شدن مدل با fit نیاز است loss function, optimizer و برخی از معیار های نظارتی مانند accuracy را تعیین کنیم به طور کلی پیکربندی training مدل را مشخص می کنیم:

```
# Specify the training configuration (optimizer, loss, metrics)
model.compile(loss="sparse_categorical_crossentropy", optimizer="sgd", metrics=["accuracy"])
```

سپس مدل را روی داده های training ، fit میکنیم.مدل را با تقسیم داده به sliceهایی به نام batch به سبب batch_size, train می کنیم در هر iteration روی کل دیتاست به تعداد epochs تکرار می کنیم.

در این بخش مجموعه ی validation را برای کنترل loss و accuracy آن در انتهای هر epoch می دهیم.

```
# Fit model on training data
# Train the the entire dataset for a given number of "epochs"
history = model.fit(X_train, y_train, batch_size=64, epochs=3, validation_data=(X_valid, y_valid))
```

سپس object ای به نام history را چاپ می کنیم که تمام گزارشات مقادیر loss و مقادیر accuracy در طول training را نشان می دهد.

```

1
2 # The returned "history" object holds a record
3 # of the loss values and metric values during training
4 print('\nhistory dict:', history.history)
5

```

و سپس مدل را با مجموعه ی test ارزیابی می کنیم و مقادیر loss و accuracy را برای مجموعه ی test گزارش می کنیم.

```

# Evaluate the model on the test data using "evaluate"
print("\n# Evaluate on test data")
results = model.evaluate(X_test, y_test, batch_size=128)
print('test loss, test acc:', results)

```

نتایج با epochs=3 عبارت است از:

```

Run: main x
43712/55000 [=====>.....] - ETA: 0s - loss: 0.2806 - accuracy: 0.9193
44608/55000 [=====>.....] - ETA: 0s - loss: 0.2793 - accuracy: 0.9199
45504/55000 [=====>.....] - ETA: 0s - loss: 0.2781 - accuracy: 0.9202
46464/55000 [=====>.....] - ETA: 0s - loss: 0.2779 - accuracy: 0.9203
47296/55000 [=====>.....] - ETA: 0s - loss: 0.2773 - accuracy: 0.9204
48192/55000 [=====>.....] - ETA: 0s - loss: 0.2760 - accuracy: 0.9208
49152/55000 [=====>.....] - ETA: 0s - loss: 0.2762 - accuracy: 0.9207
50048/55000 [=====>.....] - ETA: 0s - loss: 0.2758 - accuracy: 0.9209
50880/55000 [=====>.....] - ETA: 0s - loss: 0.2753 - accuracy: 0.9209
51776/55000 [=====>.....] - ETA: 0s - loss: 0.2744 - accuracy: 0.9211
52672/55000 [=====>.....] - ETA: 0s - loss: 0.2736 - accuracy: 0.9214
53504/55000 [=====>.....] - ETA: 0s - loss: 0.2733 - accuracy: 0.9213
54400/55000 [=====>.....] - ETA: 0s - loss: 0.2726 - accuracy: 0.9215
54976/55000 [=====>.....] - ETA: 0s - loss: 0.2725 - accuracy: 0.9216
55000/55000 [=====] - 3s 63us/sample - loss: 0.2726 - accuracy: 0.9215 - val_loss: 0.2367 - val_accuracy: 0.9344

history dict: {'loss': [0.8728365136298266, 0.33697856601259923, 0.27256456521641126], 'accuracy': [0.7653818, 0.9041455, 0.92152727], 'val_loss': [0.3699067366838455, 0.27797

# Evaluate on test data

10000/1 [=====]
test loss, test acc: [32.83017091064453, 0.9279]

Process finished with exit code 0

```

قسمت دوم)

در این قسمت می خواهیم طبقه بندی را با استفاده از یک شبکه کانولوشنی انجام دهیم. بعد از load داده ها از `tf.keras.datasets.mnist.load.data()` ، داده را برای deep learning به وسیله ی keras ، reshape می کنیم و هر نمونه ی دو بعدی به وسیله ی یک تصویر 28*28 grayscale نمایش داده می شود. سپس برای از مجموعه ی train مجموعه ی validation را می سازیم.

```

# the data, shuffled and split between train and test sets
data= tf.keras.datasets.mnist.load_data()
(X_train, Y_train), (X_test, Y_test) = data

# input image dimensions
img_rows, img_cols = 28, 28

# Preprocess the data

# Reshaping of data for deep learning using Keras
# 2 dimensions per example representing a greyscale image 28x28.
X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

print(X_train.shape)

# create a validation set
X_valid, X_train = X_train[:5000] / 255.0, X_train[5000:] / 255.0
y_valid, y_train = Y_train[:5000], Y_train[5000:]

```

سپس مدل را به وسیله ی alexnet (Keras Sequential API) می سازیم.

```

# Create a model using Keras Sequential API
model = keras.models.Sequential([
    keras.layers.Conv2D(64, 7, activation="relu", padding="same",
        input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same",),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same",),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same",),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same",),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation="softmax")
])

```

بقیه ی مراحل مانند قسمت اول است.

نتیجه نهایی در epochs=3 می شود:

```

Run: main x
54336/55000 [=====>.] - ETA: 7s - loss: 0.3086 - accuracy: 0.9124
54400/55000 [=====>.] - ETA: 6s - loss: 0.3084 - accuracy: 0.9124
54464/55000 [=====>.] - ETA: 5s - loss: 0.3085 - accuracy: 0.9124
54528/55000 [=====>.] - ETA: 5s - loss: 0.3085 - accuracy: 0.9124
54592/55000 [=====>.] - ETA: 4s - loss: 0.3086 - accuracy: 0.9123
54656/55000 [=====>.] - ETA: 3s - loss: 0.3085 - accuracy: 0.9124
54720/55000 [=====>.] - ETA: 2s - loss: 0.3085 - accuracy: 0.9124
54784/55000 [=====>.] - ETA: 2s - loss: 0.3086 - accuracy: 0.9124
54848/55000 [=====>.] - ETA: 1s - loss: 0.3087 - accuracy: 0.9124
54912/55000 [=====>.] - ETA: 0s - loss: 0.3088 - accuracy: 0.9124
54976/55000 [=====>.] - ETA: 0s - loss: 0.3086 - accuracy: 0.9124
55000/55000 [=====] - 601s 11ms/sample - loss: 0.3086 - accuracy: 0.9124 - val_loss: 0.0997 - val_accuracy: 0.9720

history dict: {'loss': [1.8361531470298766, 0.5463264453194359, 0.30858384381857784], 'accuracy': [0.3581091, 0.8321091, 0.9124], 'val_loss': [0.4419163644313812, 0.1443668800]}

# Evaluate on test data

10000/1 [=====]
test loss, test acc: [18.016370149993897, 0.9699]

Process finished with exit code 0

```

مشاهده می شود دقت در alexnet (استفاده از شبکه کانولوشنی) بالاتر از دقت در شبکه ی fully connected است و مقدار loss آن از fully connected کم تر است زیرا در alexnet به دلیل 2 بعدی بودن، اطلاعات نمونه ها یا عکس ها با دقت بالاتری نگهداری می شود. علاوه بر این ، به دلیل نداشتن افزونگی و داشتن پارامتر های کم تر می تواند تعداد لایه ی بیش تری را داشته باشد و در نتیجه دقت بالا تری را داشته باشد.

fully connected دارای افزونگی است زیرا تعداد پارامتر ها می تواند بسیار زیاد شود (تعداد perceptron در لایه ی 1 ضرب می شود سپس در تعداد perceptron در لایه دوم و ...) و این باعث ناکارآمدی این مدل می شود. همچنین fully connected به اطلاعات مکانی نمونه ها توجهی نمی کند و ورودی آن فقط یک بردار مسطح و یک بعدی است نه دو بعدی. پس دقت fully connected پایین تر و مقدار loss آن بالاتر است.

یک MLP با وزن سبک و تعداد لایه ی کم می تواند با داده های MNIST به دقت بالایی دست یابد.