توضیحات سوال اول:

سوال اول یک perceptron ساده بود و ما در حقیقت یک نورون داریم و دو داده برای train شدن. حال یک کلاس perceptron ساخته شده است که ابتدا مقادیر bias و وزن ها را بصورت رندوم ایجاد میکند. یک تابع fit هم خواهیم داشت که عملیات train شدن را برای ما انجام میدهد. ورودی های این تابع learning rate و تعداد iteration ها را به همراه داده ها هستند.

```
def fit(self, x, y, learning_rate=0.01, num_iter=100):

  (n_feature, n_inputs) = x.shape

  # Randomly initialize the weights
  self.weights = np.random.rand(n_feature)

  # Main perceptron iteration
  for i in range(num_iter):

    predict = np.dot(self.weights, x) + self.bias

    # Caculate the error
    diff = y- predict
    errors = (1 / n_inputs) * np.sum((1/2) * (diff)**2)

    # Update the weights (Avg of the changes)
    weights_diff = np.sum((diff) * x, axis = 1)
    self.weights += (1 / n_inputs) * learning_rate *  weights_diff
    self.bias += (1 / n_inputs) * learning_rate *  np.sum(diff)

    if i % 100 == 0:
      print("Iteration number {}\tBias: {}\tweights: {}     error: {}".
            format(i, self.bias, self.weights, errors/n_inputs))
```

در هر حلقه Predict را بوسیله وزن ها حساب کرده و با استفاده از error ای که بدست میاید وزن ها وbias را برای این perceptron با توجه به learning rate آپدیت خواهیم کرد در هر مرحله میانگین error ها برای تمام داده های ما در نظر گرفته میشود و در نهایت به این وزن ها خواهیم رسید.

```
Iteration number 0      Bias: -0.005    weights: [0.38737257 0.44376177]     error: 0.2765236144202956
Iteration number 100    Bias: -0.31881399106975156      weights: [-0.17519294 -0.15455267]      error: 0.06411167911126833
Iteration number 200    Bias: -0.4336800609453089       weights: [-0.38111011 -0.3735551 ]      error: 0.0356527970826144
Iteration number 300    Bias: -0.47572475743471354      weights: [-0.45648246 -0.45371708]      error: 0.03183988532159427
Iteration number 400    Bias: -0.4911144761288527       weights: [-0.48407117 -0.48305895]      error: 0.03132903264359068
Iteration number 500    Bias: -0.4967476108940049       weights: [-0.49416953 -0.49379903]      error: 0.03126058876789144
Iteration number 600    Bias: -0.4988095204080107       weights: [-0.49786586 -0.49773024]      error: 0.031251418679679246
Iteration number 700    Bias: -0.4995642459650673       weights: [-0.49921884 -0.4991692 ]      error: 0.03125019007424215
Iteration number 800    Bias: -0.4998404999303744       weights: [-0.49971407 -0.4996959 ]      error: 0.03125002546608516
Iteration number 900    Bias: -0.4999416178160817       weights: [-0.49989534 -0.49988869]      error: 0.03125000341193781
Iteration number 1000   Bias: -0.4999786302325317       weights: [-0.49996169 -0.49995926]      error: 0.03125000045713032
Iteration number 1100   Bias: -0.4999921779739812       weights: [-0.49998598 -0.49998509]      error: 0.03125000006124617
Iteration number 1200   Bias: -0.49999713688550296      weights: [-0.49999487 -0.49999454]      error: 0.03125000000820574
Iteration number 1300   Bias: -0.4999989520074974       weights: [-0.49999812 -0.499998  ]      error: 0.031250000001099405
Iteration number 1400   Bias: -0.4999996164008504       weights: [-0.49999931 -0.49999927]      error: 0.0312500000001473
Iteration number 1500   Bias: -0.499999859590305        weights: [-0.49999975 -0.49999973]      error: 0.03125000000001974
Iteration number 1600   Bias: -0.4999999486055105       weights: [-0.49999991 -0.4999999 ]      error: 0.03125000000000264
Iteration number 1700   Bias: -0.49999998118795463      weights: [-0.49999997 -0.49999996]      error: 0.031250000000000354
Iteration number 1800   Bias: -0.49999999311418286      weights: [-0.49999999 -0.49999999]      error: 0.03125000000000005
Iteration number 1900   Bias: -0.4999999997495682       weights: [-0.5 -0.5]     error: 0.03125000000000001
Iteration number 2000   Bias: -0.4999999990774406       weights: [-0.5 -0.5]     error: 0.03125000000000001
Iteration number 2100   Bias: -0.4999999996623132       weights: [-0.5 -0.5]     error: 0.03125
Iteration number 2200   Bias: -0.49999999987639576      weights: [-0.5 -0.5]     error: 0.031249999999999997
Iteration number 2300   Bias: -0.4999999999547568       weights: [-0.5 -0.5]     error: 0.03125
Iteration number 2400   Bias: -0.4999999999834397       weights: [-0.5 -0.5]     error: 0.03125
```

توضیحات سوال دوم:

ابتدا فایل را به یک آرایه از نوع نامپای تبدیل میکنیم و در ادامه برای آن که بتوانیم سوال را به درستی حل کنیم در ابتدا باید داده های ورودی خود را Normalize و Standardize کنیم. با این کار محدوده دیتا های ما بین صفر و یک خواهد شد. برای آن که بتوانیم نقاط با مقادیر 0 و یک را در یک تصوی نشان دهیم figure را به شکل زیر ساختیم.

```
data = np.genfromtxt('data.txt', delimiter=',')
data = data.astype('float')

# Normalize & Standardize
maximum = np.max(data[:, :2], axis=0)
minimum = np.min(data[:, :2], axis=0)
normalized = (data[:, :2] - minimum) / (maximum - minimum)
standardized = normalized / np.std(normalized, axis=0)

fig, (ax, err_ax) = plt.subplots(1, 2)
```

برای راحتی بیشتر داده های 0 را تبدیل به 1- میکنیم و در داخل همان کلاس سوال قبل پاس میدهیم.

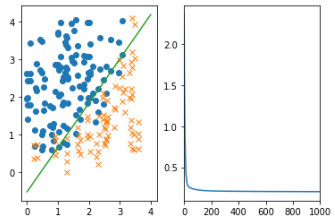
```
# Change 0s to -1
zero_data = data[data[:, 2] == 0]
zero_data[:, 2] = -1
data[data[:, 2] == 0] = zero_data
result  = data[:, 2]

p = Perceptron()

n_iter = 10000
p.fit(input_data, result, learning_rate=0.01, num_iter=n_iter)
```

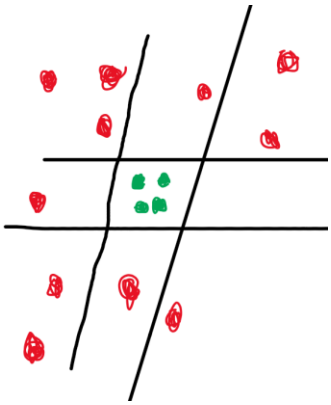
در این جا هم وزن ها را در مقادیر ضرب ماتریسی میکنیم و با بایس جمع خواهیم کرد. تنها نکته اضافی نگه داشتن error ها است تا بتوانیم در انتها با استفاده از آن نمودار را رسم کنیم.

```
Iteration number 9000   Bias: 0.2514933743239349       weights: [-0.5734038   0.48839399]     error: 0.20261776210274576
-------------
Iteration number 9100   Bias: 0.25149499726227503      weights: [-0.57340417  0.48839359]     error: 0.2026177620998083
-------------
Iteration number 9200   Bias: 0.2514964439195354       weights: [-0.57340449  0.48839323]     error: 0.20261776209747429
-------------
Iteration number 9300   Bias: 0.2514977334430976       weights: [-0.57340478  0.4883929 ]     error: 0.2026177620956198
-------------
Iteration number 9400   Bias: 0.2514988829005835       weights: [-0.57340503  0.48839262]     error: 0.20261776209414628
-------------
Iteration number 9500   Bias: 0.2514999075057558       weights: [-0.57340526  0.48839236]     error: 0.20261776209297547
-------------
Iteration number 9600   Bias: 0.25150082081988206      weights: [-0.57340546  0.48839213]     error: 0.2026177620920452
-------------
Iteration number 9700   Bias: 0.25150163493122507      weights: [-0.57340565  0.48839193]     error: 0.20261776209130603
-------------
Iteration number 9800   Bias: 0.25150236061503967      weights: [-0.57340581  0.48839175]     error: 0.20261776209071877
-------------
Iteration number 9900   Bias: 0.2515030074761885       weights: [-0.57340595  0.48839159]     error: 0.20261776209025206
-------------
```

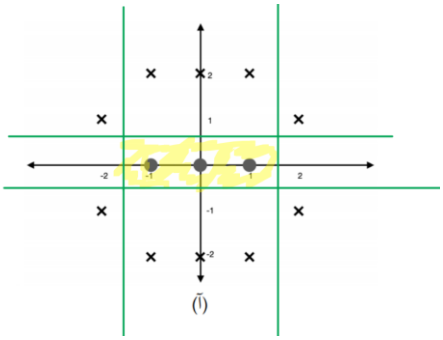


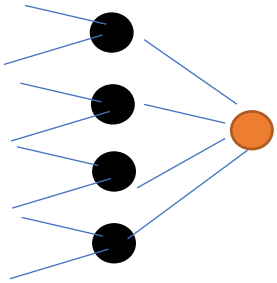
توضیحات سوال چهارم:

Madaline تعدادی واحد به نام Adaline وجود دارند که هر کدام میتواند مانند یک خط عمل کند و کلا در مدلاین ما میتوانیم چندین خط داشته باشیم و بوسیله شکلی که در بین آن ها و در اشتراک آن ها بوجود آمده است داده های خود را تفکیک کنیم. پس تفاوت با perceptron همین است. طبق نکته ای که سر کلاس هم اشاره شد و باید به آن توجه داشت مه باید ناحیه ما باید تشکیل یک چندضلعی محدب بدهد و در واقع داده های ما جدا پذیر باشند. در شکل زیر ناحیه سبز رنگ به وسیله یک convex یا چندضلعی محدب قابل پوشش است و هم چنین ناحیه ها از همدیگر جداپذیر هستند. در حقیقت هر کدام از این خط ها ادلاین های ما هستند که از and آن ها ناحیه بدست آمده است.

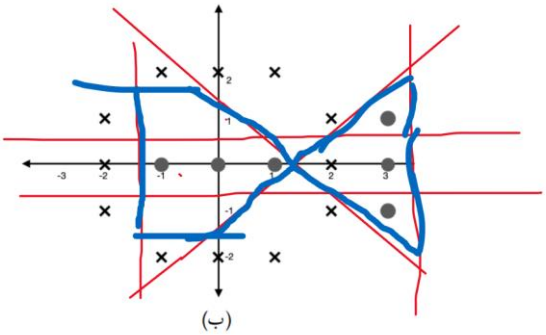


برای مورد الف میتوانیم ناحیه ای که در آن دایره ها هستند را بوسیله یک convex یا چند ضلعی محدب تفکیک نمود. در اینجا پس ما میتوانیم 4 ادلاین بصورت parallel داشته باشیم که یک مدلاین را تشکیل میدهند. و از همه آن ها به یک نرون دیگر میروند تا با هم and شوند.





در مورد ب این مطلب صادق نیست و این دو ناحیه جدا ناپذیر هستند و نمیشود بوسیله یک مدلاین آن ها را از یک دیگر تفکیک کرد.



توضیحات سوال چهارم:

بیشتر سوال چهارم کار با keras بود وگرنه از بین بقیه سوالات پیچیدگی کمتری داشت.داده های mnist را لود کردیم. عکس ها 28 در 28 هستند و هر پیکسل بین 0 تا 255 است و دوباره اینجا از عملیات scaling استفاده شد تا نتایج درست به دست بیایند و داده های ما normalized باشند.

```
# Digits should a number from zero to nine
n_classes = 10

# Scaling, Normalizing and Standarizing
# Scale
x_train, x_test = scale_image(x_train), scale_image(x_test)
x_train = x_train.reshape(60000, 784)

# Reshape
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
```

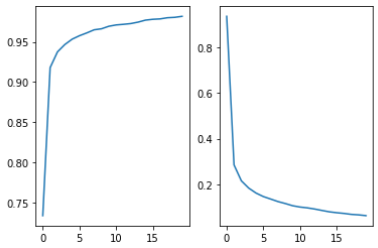
در keras به راحتی میتوانیم در داخل یک مدل لایه های خود را قرار دهیم. برای این سوال من یک مدل با سه لایه hidden و یک لایه پایانی درست کردم. لایه از نوع Dense چیزی بود که تا اینجا بلد هستیم و در هر لایه activation function را مشخص خواهیم کرد که در سه لایه اول relu هست که زیر 0 صفر و بالای صفرمقدار خود را دارد. لایه آخر هم که softmax است.

```
# The model
model = keras.Sequential(
    [
        layers.Dense(units=32, input_shape=(784, ), activation='relu'),
        layers.Dense(units=64, activation='relu'),
        layers.Dense(units=32, activation='relu'),
        layers.Dense(n_classes, activation="softmax"),
    ]
)
```

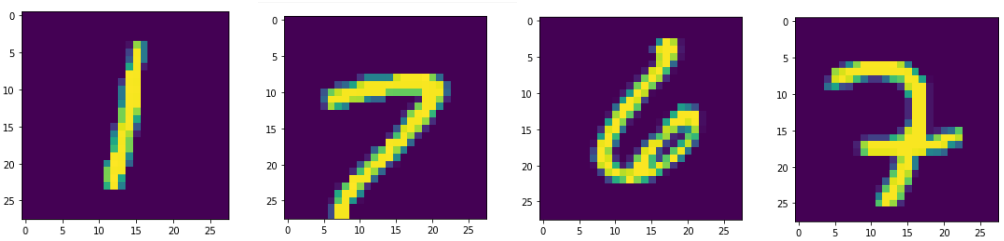
Epoch همان تعداد iteration های ماست و batch هم همان اندازه دسته دسته دیتا هایی است که در هر iteration بررسی میشوند. در اینجا mini batch داریم و در هر مرحله 500 مورد بررسی میشود. آن یک دهم هم برای validation هست که خوب حواسمون باشه overfit نشیم.

```
# Training
history = History()
epochs=20
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
model.fit(x_train, y_train, batch_size=500, epochs=epochs, validation_split=0.1, verbose=2, callbacks=[history])
```

```
Epoch 14/20
108/108 - 1s - loss: 0.0853 - accuracy: 0.9745 - val_loss: 0.1099 - val_accuracy: 0.9708
Epoch 15/20
108/108 - 1s - loss: 0.0793 - accuracy: 0.9770 - val_loss: 0.1064 - val_accuracy: 0.9710
Epoch 16/20
108/108 - 1s - loss: 0.0754 - accuracy: 0.9780 - val_loss: 0.1096 - val_accuracy: 0.9710
Epoch 17/20
108/108 - 1s - loss: 0.0723 - accuracy: 0.9785 - val_loss: 0.1126 - val_accuracy: 0.9703
Epoch 18/20
108/108 - 1s - loss: 0.0681 - accuracy: 0.9799 - val_loss: 0.1044 - val_accuracy: 0.9737
Epoch 19/20
108/108 - 1s - loss: 0.0659 - accuracy: 0.9804 - val_loss: 0.1053 - val_accuracy: 0.9727
Epoch 20/20
108/108 - 1s - loss: 0.0626 - accuracy: 0.9817 - val_loss: 0.1028 - val_accuracy: 0.9737
```



توضیحات سوال پنجم:



عکس های ما دوباره همان عکس ها 28 در 28 هستند دوباره آن ها را Normalize میکنیم و scale میکنیم. یک مدل 2 لایه در نظر گرفته شده است که لایه اول از تابع relu و لایه آخر از تابع softmax استفاده میکند و به ترتیب تعداد 130 و 10 تا نرون دارند. (دقیقا مانند سوال قبل) حال دیتا های train را به شکل (784, 60000) تبدیل کرده و همچنین نتایج را به شکل (1, 600000) تبدیل کرده ایم. این مقادیر ورودی شبکه ای میشود که ما بصورت scratch با استفاده از numpy طراحی کرده ایم. در این شبکه و در هر iteration خطا بروی همه داده های ترین سنجیده میشود و به عبارتی دیگر ما از batch gradient descent استفاده میکنیم.

در ابتدا وزن ها و بایس ها را بصورت رندوم میسازیم و شکل آن ها را متناسب میکنیم. 0.01 برای کمتر کردن مقادیر بین 0 تا 100 تابع رندوم است.

```
# Generate random weights
# 1
# First layer has 130 neurons with 784 inputs.
weights_layer1 = np.random.rand(130 * 784) * 0.01
weights_layer1 = weights_layer1.reshape(130, 784)
bias_layer1 = np.random.rand(130) * 0.01
bias_layer1 = bias_layer1.reshape(130, 1)

# 2
# Second layer has 10, too with 130 inputs.
# Activations2 = weights_layer2 dot Activations1 + bias_layer2
weights_layer2 = np.random.rand(10 * 130) * 0.01
weights_layer2 = weights_layer2.reshape(10, 130)
bias_layer2 = np.random.rand(10) * 0.01
bias_layer2 = bias_layer2.reshape(10, 1)
```

سپس در هر iteration ما ابتدا عملیات forward propagation را انجام خواهیم داد. با توجه به ابعاد ورودی کافی است وزن را در داده ها ضرب ماتریسی کرد و با بایس جمع کرد.

```
# Forward propagation
# Output is (10 * 1) array showing the activation to be number  0 <= i < 9
activations1, z1 = relu(np.dot(weights_layer1, x) + bias_layer1)
output, z2 = softmax(np.dot(weights_layer2, activations1) + bias_layer2)
```

در مرحله بعدی باید خطا را حساب کنیم که من از همان MSE برای سنجش میزان ارور استفاده کرده ام.

```
    # Calculate the error
    diff = (output - y)
    loss = np.sum(np.square(diff) / 2)
    cost = loss / 60000
    epoch_loss.append(cost)
```

در ادامه باید عملیات back propagation را انجام بدهیم. در ابتدا کمی محاسبات لازم است.

☐ forward Propagation

w_i	وزن لایه i ام
x_i	داده‌ی train
b_i	بایاس لایه i ام

$$z = \sum_{i=1}^{n} W_i X_i + b_i \quad \text{(linear activation)}$$

n تعداد لایه

$$a = \sigma(z) \quad \text{(activation function)}$$

تابع Softmax $\longrightarrow$ $\dfrac{e^{z_i}}{\sum_j e^{z_j}} = \sigma(z)$

تابع Relu $\longrightarrow$ $\max\{0, z\} = \sigma(z)$

تابع MSE $\longrightarrow$ $\dfrac{\sum_{i}^{n} (Y_i - Y_i^P)^2}{n}$

برای محاسبه Cost , loss

درون هر iteration (حلقه)

برای یک شبکه که دارای L لایه است و m داده دارد :

$Z^{[1]} = W^{[1]} A^{[0]} + b^{[1]} \Rightarrow Z^{[L]} = W^{[L]} A^{[L-1]} + b^{[L]}$

(ورودی — یعنی لایه اول)

activation function $\Rightarrow A^{[L]} = \sigma(Z^{[L]})$

Mse cost function $\Rightarrow \frac{1}{m} \sum \left(\frac{1}{2} (y - y^p)^2 \right)$

☐ Back propagation

$da \Longrightarrow$ مشتق cost function = diff = $y - y^p$

$$\frac{\partial C}{\partial W} = \frac{\partial z}{\partial w}^{③} \frac{\partial a}{\partial z}^{②} \frac{\partial C}{\partial a}^{①}$$

مشتق relu $\longrightarrow$ برای آسان‌تر برابر صفر و گرنه برابر ۱

مشتق Softmax $\longrightarrow$ $Softmax(dz) \times (1 - Softmax()) \times da$

$\Rightarrow$ da = مشتق costfunction لایه آخر

$\Rightarrow$ $dz^{[L]} = dA^{[L]} \times$ مشتق تابع $(z^{[L]})$
activation

$\Rightarrow$ $dw^{[L]} = \frac{1}{m} dz^{[L]} \times A^{[L]}$

$\Rightarrow$ $db^{[L]} = \frac{1}{m} dz^{[L]}$

$\Rightarrow$ $da^{[L-1]}$ بجز لایه آخر $= W^{[L-1]} \times dz^{[L]}$

برای اکتیو کردن وزن کار ساده‌ایست اگر η را learning rate در نظر بگیریم...

$$W^{[L]} = W^{[L]} - \eta\, dW^{[L]}$$

$$b^{[L]} = b^{[L]} - \eta\, db^{[L]}$$

```
# Back propagation
da2 = diff

dz2 = relu_diff(da2, z2)
da1 = np.dot(weights_layer2.T, dz2)
m = da1.shape[1]
dw2 = (1 / m) * np.dot(dz2, da1.T)
db2 = np.sum(dz2, axis=1, keepdims=True) / m

dz1 = relu_diff(da1, z1)
da0 = np.dot(weights_layer1.T, dz1)
m = da0.shape[1]
dw1 = (1 / m) * np.dot(dz1, da0.T)
db1 = np.sum(dz1, axis=1, keepdims=True) / m
```

همانطوری که محاسبات ریاضی انجام شد به شکل زیر آن ها را پیاده سازی میکنیم. در محاسبه کردن da ها بدلیل این که با ابعاد ابتدایی نمیتوانیم ضرب ماتریسی انجام بدیم مجبور هستیم که ترانهاده ماتریس وزن آن لایه را در dz ضرب کنیم تا ابعاد با هم ضرب پذیر باشند. در انتها نیز با توجه به learning rate و گرادیانت وزن در جهت مخالف وزن ها را تغییر می دهیم تا خطای ما مینیمم شود.

```
# Update the weights
weights_layer1 -= learning_rate * dw1
bias_layer1 -= learning_rate * db1

weights_layer2 -= learning_rate * dw2
bias_layer2 -= learning_rate * db2

weights_layer3 -= learning_rate * dw3
bias_layer3 -= learning_rate * db3
```