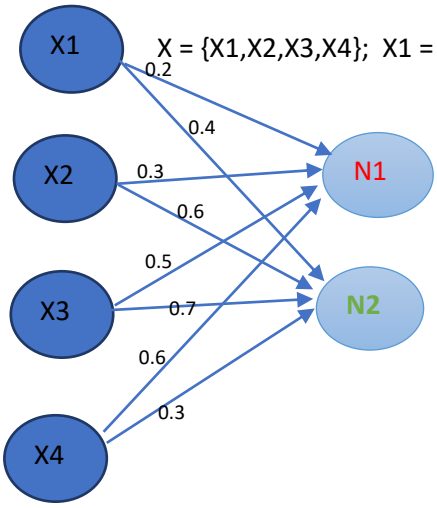
تمرین سوم هوش محاسباتی

بنفشه قلی نژاد 98522328

سوال 1)

مراحل مدل kohenenبه شرح زیر می باشد.

1- Initialization vectors
2- Argmin||X – Wj||
3- Cooperation : Gaussian function
4- Adaptation : $W(t+1) = Wt + \alpha(X - Wi)$



X = {X1,X2,X3,X4}; X1 = [1 1 0 0] , X2 = [0 0 0 1] , X3 = [0 1 1 0] , X4 = [1 0 0 0], $\alpha = 0.5$

مدل خواسته شده به شرح زیر است:

For data 1 ➔ X1 = [1 1 0 0]

- ✓ **Initialize random weights**

Wji = [W11 , W12 , W13 , W14 , W21 , W22 , W23 , W24] ➔ Wji ={0.2 , 0.3 , 0.5 , 0.6 , 0.4, 0.6 , 0.7, 0.3}

- ✓ **Competition**

d1=$\sqrt{(0.2-1)^2 + (0.3-1)^2 + (0.5)^2 + (0.6)^2} = \sqrt{0.64 + 0.49 + 0.25 + 0.36} = \sqrt{1.74}$

d2 = $\sqrt{(0.4-1)^2 + (0.6-1)^2 + (0.7)^2 + (0.3)^2} = \sqrt{0.36 + 0.16 + 0.49 + 0.09} = \sqrt{1.1}$

min(d1 , d2) = d2 ➔ **N2 wins**

- ✓ **Cooperation and adaption**

W2i = [0.4 , 0.6 , 0.7 , 0.3] + 0.5([1, 1, 0 ,0] – [0.4, 0.6, 0.7 ,0.3]) = [0.4 , 0.6 , 0.7 , 0.3] +0.5 ([0.6 , 0.4 ,-0.7 , -0.3]) == [0.4 , 0.6 , 0.7 , 0.3] + [0.3 , 0.2 , -0.35 ,-0.15] = **[0.7 , 0.8 , 0.35 ,0.15]**

For data 1 ➔ X2 = [0 0 0 1]

- ✓ **Updated weights**

Wji=[W11 , W12 , W13 , W14 , W21 , W22 , W23 , W24] ➔ Wji={0.2 , 0.3 , 0.5 , 0.6, 0.7 , 0.8 , 0.35 ,0.15}

- ✓ **Competition**

d1=$\sqrt{(0.2)^2 + (0.3)^2 + (0.5)^2 + (0.6 - 1)^2} = \sqrt{0.04 + 0.09 + 0.25 + 0.16} = \sqrt{0.54}$

d2 = $\sqrt{(0.7)^2 + (0.8)^2 + (0.35)^2 + (0.15 - 1)^2} = \sqrt{0.49 + 0.64 + 1.225 + 0.7225} = \sqrt{3.0775}$

min(d1 , d2) = d1 ➔ **N1 wins**

- ✓ **Cooperation and adaption**

W1i = [0.2 , 0.3 , 0.5 , 0.6] + 0.5([0,0,0,1] – [0.2 , 0.3 , 0.5 , 0.6]) = [0.2 , 0.3 , 0.5 , 0.6] + 0.5 ([-0.2, -0.3,-0.5 , 0.4]) = [0.2 , 0.3 , 0.5 , 0.6] + [-0.1 , -0.15 , -0.25 , 0.2] = **[0.1 , 0.15 , 0.25,0.8]**

- ❖ Current weight : **Wji = [0.1 , 0.15 , 0.25 , 0.8 , 0.7 ,0.8 , 0.35 , 0.15]**

توضیح : در ادامه ی کار ، همین روند بالا برای تمامی داده ها و در تعداد دوره های مشخص تکرار می شود یعنی کوچک ترین فاصله هر نقطه داده از وزن های هر کدوم از نورون هایی که میخواهیم به آن مپ کنیم و نورون برنده نورون با کم ترین فاصله است. سپس برای اپدیت کردن وزن های نورون برنده، طبق فرمول بالا آن ها را اپدیت می کنیم. این کار را تا جایی ادامه می دهیم که اختلاف وزن ها یعنی ΔW به کم ترین مقدار خود و مقدار خواسته شده ی مابرسد. انقدر این عمل را تکرار می کنیم که هر یک از نورون های مپ شده این 4 داده را به دو دسته مختلف گروه بندی می کنند.

https://www.geeksforgeeks.org/self-organising-maps-kohonen-maps/

https://www.youtube.com/watch?v=9ZhwKv_bUx8

سوال 2)

ابتدا ماتریس وزن رادر مدل محاسبه می کنیم.

$$\sum_{k=1}^{p} X_i^k X_j^k = \text{Wij (where Wij is 0 for i == j)}$$

X1 = [1,1,1,1] , X2 = [-1,-1,-1,-1] , X3 = [-1,-1,1,1] , X4=[1,1,-1,-1]

1- Initialize matrix :

0	1 + 1 + 1 +1 = 4	1 + 1 +-1 +-1 = 0	1 + 1+=1 +-1=0
1+1+1+1 = 4	0	1 + 1 + -1 +-1=0	1+1+-1+-1=0
1+1+-1 + -1 = 0	1 + 1 + -1 + -1 = 0	0	1 + 1 + 1 + 1= 4
1+1+-1+-1 = 0	1+1+-1+-1=0	1+1+1+1=4	0

Final weight matrix :

0	4	0	0
4	0	0	0
0	0	0	4
0	0	4	0

حال پایداری ورودی ها را تعیین می کنیم.

$\sum XiWij$, Threshold = 0

R=0	X1(1,1,1,1)	X2(-1,-1,-1,-1)	X3(-1,-1,1,1)	X4(1.1,-1,-1)
R= 1	X1(1,1,1,1)	X2(-1,-1,-1,-1)	(-1,-1,1,1)	(1,1,-1,-1)
R=2	X1(1,1,1,1)	X2(-1,-1,-1,-1)	(-1,-1,1,1)	(1,1,-1,-1)

در این شبکه به علت 0و1 بودن تنهاعلامت اهمیت دارد و همین طور که مشخص است شبکه به حالت پایدار می رسد چرا که علامت ورودی ها را به خوبی حفظ کرده. با توجه به ماتریس وزن‌ها و بررسی پایدار بودن شبکه، لیست داده‌شده در صورت سوال با وزن‌های بالا قابل ذخیره‌سازی است.

https://www.geeksforgeeks.org/hopfield-neural-network/

سوال 3)

برای ساختن یک MLPمناسب، ابتدا شکل مدل را تعیین می نماییم. از آن جایی که میخواهیم y=X^2را تخمین بزنیم، ورودی 1 نورون دارد که مقدار Xاست. برای این تابع یک مدل با دو لایه ی میانی و هر کدام با 16 نورون را تعیین می نماییم.

همچنین learning_rateرا 0.01 (برای کوچک کردن قدم ها مقدار کم تری را انتخاب می کنیم)

اکتیویشن فانکشن را نیز RelU برای لایه های میانی در نظر می گیریم.

شکل مدل :

Input → hidden layer 1(16 neoron) → hidden layer2(16 neoron) → output

W1 = weight matrix(16*1)

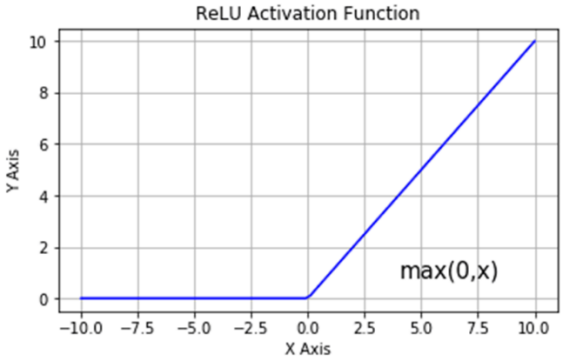
W2(hidden layer1) = Weight matrix(16*16)

W3(hidden layer 2) = Weight Matrix (1 * 16)

ابتدا برای راحتی کار فانکشن های مورد نیاز خود را تعریف می کنیم. از activation function RELU استفاده می کنیم. از آن جایی که در مرحله backpropagationبه مشتق تابع نیز نیاز داریم . آن ها را جدا از کلاس تعریف می نماییم.

این تابع در نهایت ماکسیمم جواب بین 0 تا Xرا مشخص می کند و مدل را در لایه های میانی از حالت خطی خارج می کند.

```
def relu(Z):
    return np.maximum(0,Z)
## RelU backward
def relu_backward(Z):
    return 1.*(Z>0)
```



https://builtin.com/machine-learning/relu-activation-function#:~:text=ReLU%20Activation%20Function%3F-,A%20rectified%20linear%20unit%20(ReLU)%20is%20an%20activation%20function%20that,activation%20functions%20in%20deep%20learning

https://stats.stackexchange.com/questions/333394/what-is-the-derivative-of-the-relu-activation-function

همچنین تابع خطای خود را نیز تعریف می کنیم که ساده ترین آن ها تابع MSE است.

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

```
def mse_loss(Y_hat, Y_true):
    return np.mean((Y_hat - Y_true) ** 2)
```

https://www.digitalocean.com/community/tutorials/loss-functions-in-python

حال برای طراحی MLP یک کلاس میسازیم که ورودی آن اندازه لایه های ورودی، میانی و پایان است و فانکشن های لازم برای اموزش یعنی forward propagation, backward propagation , train را تعریف می نماییم.

initialization

در وحله اول و در کانستراکتور و initializationورودی های لازم را تعریف می کنیم. ورودی شامل وزن های لایه ها و ماتریس bias است.

```
class MLP:
    def __init__(self, input_layer, hidden_layer1,
hidden_layer2,output_layer):

        self.W1 = np.random.randn(hidden_layer1, input_layer)
        self.b1 = np.zeros((hidden_layer1, 1))

        self.W2 = np.random.randn(hidden_layer2, hidden_layer1)
        self.b2 = np.zeros((hidden_layer2, 1))

        self.W3 = np.random.randn(output_layer, hidden_layer2)
        self.b3 = np.zeros((output_layer, 1))
```

forward propagation:

همان طور که میدانیم، فرمول forward propagation :

$$A = \sigma(\sum WiXi + b \ )$$

```
def forward(self, X):
      #Wi.xi + b
        self.Z1 = np.dot(self.W1, X) + self.b1
        self.A1 = relu(self.Z1)
        #layer 2
        self.Z2 = np.dot(self.W2, self.A1) + self.b2
        self.A2 = relu(self.Z2)
        #output layer
        self.Z3 = np.dot(self.W3, self.A2) + self.b3
        self.A3 = self.Z3
        return self.A3
```

Back propagation and derivations:

همان طور که در اسلاید های درسی برای backpropagation توضیح داده شد، طبق قاعده ی delta_rule ، برای لایه ی اخر و لایه های میانی از دو فرمول متفاوت استفاده می شود.

$$\delta_j = \begin{cases} \varphi'(v_j)(d_j - y_j) & \text{IF j output node} \\ \varphi'(v_j) \sum_{k \text{ of next layer}} \delta_k w_{kj} & \text{IF j hidden node} \end{cases}$$

برای محاسبه ی مشتق و derivation وزن ها ، از فرمول های زیر استفاده می کنیم.

Summary of gradient descent

$$dz^{[2]} = a^{[2]} - y \qquad dZ^{[2]} = A^{[2]} - Y$$

$$dW^{[2]} = dz^{[2]} a^{[1]T} \qquad dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

$$db^{[2]} = dz^{[2]} \qquad db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$$

$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]}) \qquad dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$$

$$dW^{[1]} = dz^{[1]} x^T$$

$$db^{[1]} = dz^{[1]}$$

Andrew Ng

در نتیجه :

```
def backward(self, X, Y, learning_rate):
        m = X.shape[1]
        dZ3 = 2 * (self.A3 - Y)
        dW3 = (1 / m) * np.dot(dZ3, self.A2.T)
        db3 = (1 / m) * np.sum(dZ3, axis=1, keepdims=True)

        dZ2 = np.dot(self.W3.T, dZ3) *relu_backward(self.Z2)
        dW2 = (1 / m) * np.dot(dZ2, self.A1.T)
        db2 = (1 / m) * np.sum(dZ2, axis=1, keepdims=True)

        dZ1 = np.dot(self.W2.T, dZ2) *relu_backward(self.Z1)
        dW1 = (1 / m) * np.dot(dZ1, X.T)
        db1 = (1 / m) * np.sum(dZ1, axis=1, keepdims=True)
```

منبع : (Deep learning.AI Andrew NG course) .Neural Networks and Deep Learning

Gradient Descent updating weights:

$$w_{new} = w - \alpha * \frac{\delta L}{\delta w}$$

$$b_{new} = b - \alpha * \frac{\delta L}{\delta b}$$

محاسبه اپدیت کردن ماتریس وزن ها طبق فرمول:

```
# Update with gradiant descent formula
        self.W3 -= learning_rate * dW3
        self.b3 -= learning_rate * db3
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
```

training:

در این قسمت با مشخص کردن training_rateو تعداد دور های آموزش ، به ترتیب مراحل forward , backward را در هر دوره اموزش پیاده سازی می کنیم.

```
  def train(self, X, Y, num_iterations, learning_rate):
        for i in range(num_iterations):
            output = self.forward(X)
            self.backward(X, Y, learning_rate)
            if i % 100 == 0:
                loss = mse_loss(output, Y)
                print(f"Epoch {i}, Loss: {loss}")
```

❋مشخص کردن داده برای Train:

همان طور که گفته شد، برای داده آموزشی از بازه (3,3-) استفاده می کنیم.

```
X_train = np.arange(-3, 3, 0.1).reshape(1, -1)
Y_train = X_train ** 2
```

حال مدل را طبق خواسته تعریف و آموزش میدهیم.

```
mlp = MLP(input_layer=1, hidden_layer1=16,
hidden_layer2=16,output_layer=1)
mlp.train(X_train, Y_train, num_iterations=2000, learning_rate=0.01)
```

مشاهده نتیجه :

```
Epoch 0, Loss: 124.68484296671338
Epoch 100, Loss: 0.15543878656076318
Epoch 200, Loss: 0.06471937060189446
Epoch 300, Loss: 0.03789536330559897
Epoch 400, Loss: 0.02529342658839911
Epoch 500, Loss: 0.018253385953440598
Epoch 600, Loss: 0.013671811665460738
Epoch 700, Loss: 0.011658049631111566
Epoch 800, Loss: 0.009582936601996939
Epoch 900, Loss: 0.008697920662745644
Epoch 1000, Loss: 0.00811432210578126
Epoch 1100, Loss: 0.0076069844784394805
Epoch 1200, Loss: 0.007230366844426933
Epoch 1300, Loss: 0.006944248314630558
Epoch 1400, Loss: 0.006709624494993129
Epoch 1500, Loss: 0.0065284937153185195
Epoch 1600, Loss: 0.005895976158942666
Epoch 1700, Loss: 0.005644526569116663
Epoch 1800, Loss: 0.005486009036034161
Epoch 1900, Loss: 0.0052541286211290755
```

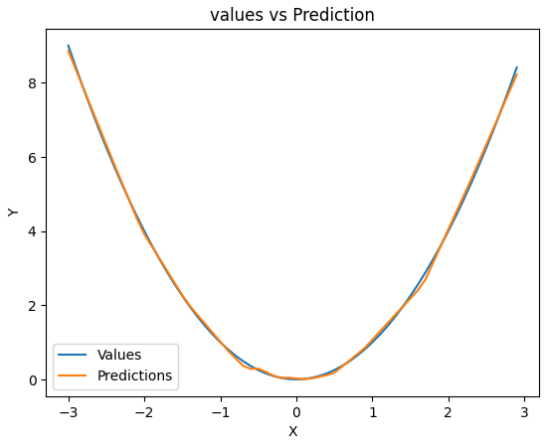
همان طور که مشاهده می شود، به مرور زمان میزان خطا در هر 100 دوره اموزشی کم تر می شود و در دوره ی اخر به کم تر از 0.01 می رسد.

مقایسه نتایج پیش بینی شده و نتایج به دست آمده در بازه ی خواسته شده:

همان طور که مشاهده میشود ، نتیجه به دست امده تا حدودی به نتیجه خواسته شده نزدیک است و مدل با تقریب خیلی خوبی مدل سهمی ساده را یاد گرفته است.

```
Predictions:
[[8.85271961e+00 8.34781533e+00 7.84291106e+00 7.33800679e+00
  6.83310251e+00 6.32819824e+00 5.82329397e+00 5.33074404e+00
  4.83826215e+00 4.34578025e+00 3.91301945e+00 3.58812843e+00
  3.25713544e+00 2.92614246e+00 2.59514948e+00 2.26415650e+00
  1.97606392e+00 1.73592371e+00 1.49578350e+00 1.25564329e+00
  1.01550308e+00 7.75362872e-01 5.69792332e-01 3.64957482e-01
  2.81089850e-01 2.90927138e-01 1.97237058e-01 1.00476450e-01
  3.59941591e-02 4.67173348e-02 2.43295794e-02 2.72588518e-03
  2.86568921e-02 6.72338626e-02 1.14388592e-01 1.76677919e-01
  3.43525080e-01 5.10372241e-01 6.77219402e-01 8.48762373e-01
  1.07119240e+00 1.29362242e+00 1.51605245e+00 1.73848247e+00
  1.96091250e+00 2.18334252e+00 2.40577255e+00 2.70048378e+00
  3.14947249e+00 3.59846121e+00 4.04744992e+00 4.49643863e+00
  4.94542735e+00 5.39651410e+00 5.86702304e+00 6.33753198e+00
  6.80804092e+00 7.27854987e+00 7.74905881e+00 8.21956775e+00]]
Actual values:
[[9.00000000e+00 8.41000000e+00 7.84000000e+00 7.29000000e+00
  6.76000000e+00 6.25000000e+00 5.76000000e+00 5.29000000e+00
  4.84000000e+00 4.41000000e+00 4.00000000e+00 3.61000000e+00
  3.24000000e+00 2.89000000e+00 2.56000000e+00 2.25000000e+00
  1.96000000e+00 1.69000000e+00 1.44000000e+00 1.21000000e+00
  1.00000000e+00 8.10000000e-01 6.40000000e-01 4.90000000e-01
  3.60000000e-01 2.50000000e-01 1.60000000e-01 9.00000000e-02
  4.00000000e-02 1.00000000e-02 7.09974815e-30 1.00000000e-02
  4.00000000e-02 9.00000000e-02 1.60000000e-01 2.50000000e-01
  3.60000000e-01 4.90000000e-01 6.40000000e-01 8.10000000e-01
  1.00000000e+00 1.21000000e+00 1.44000000e+00 1.69000000e+00
  1.96000000e+00 2.25000000e+00 2.56000000e+00 2.89000000e+00
  3.24000000e+00 3.61000000e+00 4.00000000e+00 4.41000000e+00
  4.84000000e+00 5.29000000e+00 5.76000000e+00 6.25000000e+00
  6.76000000e+00 7.29000000e+00 7.84000000e+00 8.41000000e+00]]
```

برای مقایسه، می توان نتیجه را بر روی نمودار مشاهده کرد. همان طور که مشاهده می شود، با تقریب خوب اما با کمی خطا، شکل کلی سهمی در این بازه به خوبی اموزش داده شده است.



سوال 4)

برای طراحی شبکه هاپفیلید، ابتدا باید ماتریس وزن را تعیین کنیم. همان طور که مشخص است، حالت مینیمم انرژی 111100 را از 010000را شناسایی کند. در ابتدا برای ماتریس وزن ها از یک شبکه 6*6 برای محاسبه پترن خواسته شده استفاده می کنیم.

ابتدا ماتریس وزن را با فرمول $\sum_{k=1}^{p} X_i^k X_j^k$ محاسبه میکنم.

کد زده شده برای ماتریس به صورت زیر است:

Np.outer برای هر ارایه np ایندکس های i ام و jام را در هم ضرب میکند. (همانند فرمول بالا)

https://numpy.org/doc/stable/reference/generated/numpy.outer.html

```
weight_matrix = np.zeros((input_size, input_size))
for binary in binary_sequences:
    binary = np.array(binary_sequence)
    weight_matrix += np.outer(binary, binary)
np.fill_diagonal(weight_matrix, 0)
```

Weight Matrix:

0	1	1	1	0	0
1	0	1	1	0	0
1	1	0	1	0	0
1	1	1	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

در مرحله دوم، شبکه را میسازیم که شامل 6 نورون است.

این تابع برای پیاده سازی به روز رسانی در هر مرحله از جدول محاسبات است. به این صورت که از فرمول $\sum XiWij$ استفاده می کند و مانریس وزن را در پترن خواسته شده محاسبه می کند و با کمکnp.sign (ترشولد 0) بیت 0و1 بودن آن را تعیین می کند.

```
def update(pattern, matrix):
    sum = np.dot(matrix, pattern)
    updated = np.sign(sum)
    return updated
```

این تابع همگرایی را در نظر میگیزد و هر استیت را با استیت قبلی خود مقایسه می کند تا به وضعیت پایدار برسد.

```
def convergence(current, previous):
    return np.array_equal(current, previous )
```

در نهایت در مدل hopfield، استیت قبلی را با استیت کنونی مقایسه می کند و تا زمانی که به حالت پایدار نرسد ان را ادامه میدهد. در نهایت در این مدل با ورودی اولیه سوال یعنی 01000 به خروجی 11100 میرسیم.

```
def run_hopfield_network(pattern, weight_matrix):
    current = pattern.copy()
    previous = np.zeros_like(pattern)
    while not convergence(current, previous):
        previous = current.copy()
        current = update(current, weight_matrix)
    return current
```

```
Result: [1. 1. 1. 1. 0. 0.]
```

ارائه ی جدول محاسبات:

(برای هر بیت، هر ردیف متناظر ماتریس وزن در پترن ضرب می شود و مجموع آن محاسبه میشود.)

	0	1	0	0	0	0
Round1	Sign(1)=1	Sign(0) = 0	Sign(1) =1	Sign(1) = 1	Sign(0) = 0	Sign(0) = 0
Round2	Sign(2) =1	Sign(3) = 1	Sign(2) =1	Sign(2) =1	Sign(0) = 0	Sign(0) = 0
Round3	sign(3) = 1	Sign(3) = 1	Sign(3) = 1	Sign(3)=1	Sign(0)=0	Sign(0) = 0

با توجه به جدول محاسبات نتیجه به پترن خواسته شده همگرا می شود.

سوال 5)

مسئله ی دوره گرد یک مسئله ی NPhard است و به صورت مستقیم الگوریتمی برای آن وجود ندارد. به طور کلی از یک نقطه شروع می کند و بهینه ترین دور ممکن را در شهر های خواسته شده بزند.

به نظر من بهترین انتخاب ممکن برای پیدا کردن کم ترین دور ممکن استفاده از شبکه ی SOM است. این شبکه نقطه های گراف را در یک صفحه دو بعدی مپ می کند و می توان آن را طوری آموزش داد که کم هزینه ترین دور ممکن را بیابد. از آن جایی که در خود این مسئله نیز برای برنده شدن هر نود بهینه بودن از لحاظ فاصله ی اقلیدسی را بررسی می کند می تواند انتخاب مناسبی باشد.

در این مسئله ، تعداد نورون های مپ به اندازه تعداد شهر های گراف و یا بیشتر است. نحوه ی الگوریتم به شرح زیر است:

1- Initialization :

در این مرحله به نورون های اولیه مقدار اولیه لحاظ میکنیم تا مکان اولیه آن ها مشخص شود.

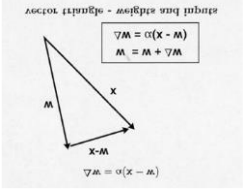
2- Competition

در این مرحله فاصله نورون ها از نقاط ورودی بررسی می شود و هر نورون نزدیک ترین شهر به خودش را انتخاب می کند.

3- Cooperation : برای محاسبه ی همسایگی نورون ها و پیدا کردن نورون های نردیک و در همسایگی می توان از فرمول گاوسی و یک sigma مشخص استفاده کرد و برای دقیق تر کردن نتیجه بعد از هر مرحله شعاع همسایگی را کاهش داد

4- $$h_{j,i(x)} = e^{\frac{-d_{j,i}^2}{2\sigma^2}}$$

5- Adaptation : در این مرحله برای به روز رسانی وزن ها استفاده می شود و اختلاف هر نقطه از ورودی تعیین می شود و ضرب در learning_rate شعاع همسایگی مشخص می شود.



یک مدل فرضی برای این سوال:

1- Initialization part:

در این مرحله یک مپ در نظر گرفته و برای هر نورون به عنوان وزن مختصات های متفاوت را تعیین می نماییم.

```
x = np.random.rand(100)
y = np.random.rand(100)
```

```
som = np.column_stack((x, y)) * 1000
```

سپس هایپر پارامتر ها را تعیین می نماییم.

شعاع گاوسی را ابتدا یک عدد رندوم و سیگما را 0.5در نظر میگیریم و برای learning rateیک مقدار تعیین می کنیم.

```
iterations = 600
sigma = 30
learning_rate= 0.3
```

یکی از توابع لارم تابع گاوسی است که طبق فرمول آن را تعریف مینماییم.

```
#define Gaussian function
radius = np.random.rand()
def Gaussian_neighborhood(radius, r, sigma):
  return  np.exp(-(np.abs(radius - r) ** 2) / (2 * sigma ** 2))
```

در مرحله ی بعد، طبق الگوریتم توضیح داده شده، (دیتای در نظر گرفته شده بک تابع شامل مختصات شهر های مختلف است که باید بررسی شود.(در کد داده شده این لیست خالی است)

```
for i in range(iterations):

  #decrease sigma value in each round in order to be more accurate in
farther iterations
    sigma -= 0.05
    for j in range(len(Data)):
        #Competition of neorons for each data
        r_min = np.argmin(np.linalg.norm(som - Data[j], axis=1))
        #Adaptation and update weight values
        for k in range(len(som)):
            som[k] += learning_rate * Gaussian_neighborhood(k, r_min,
sigma) * (Data[j] - som[k])
```

همان طور که مشخص است، طبق توضیح در مرحله اول نزدیک ترین نورون انتخاب می شود. سپس بر اساس کم ترین فاصله ، تابع همسایگی برای نورون برنده را در نرخ یادگیری و اختلاف نقطه خواسته شدهو مختصات اولیه به دست می اوریم و طبق آن وزن نورون ها را اپدیت میکنیم. انجام این مراحل به مرور زمان باعث میشود که در دور های اخر ، کم ترین دور ممکن برای هر نورون شناحته میشود.

https://cse22-iiith.vlabs.ac.in/exp/self-organizing-maps/theory.html