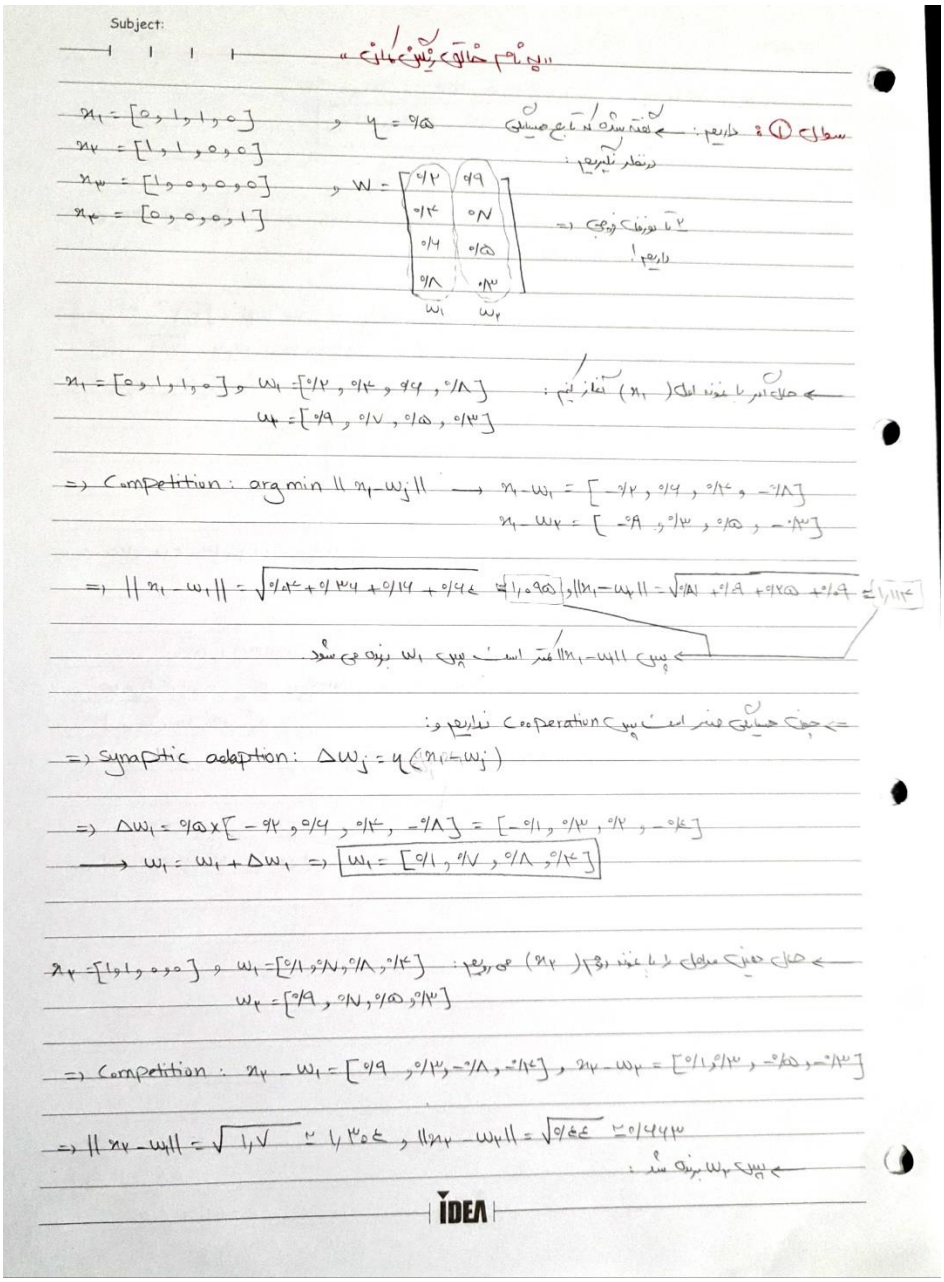
به نام خالق رنگین کمان

ستاره باباجانی - 99521109

سوال 1:



Subject:

$\Rightarrow \Delta w_2 = ۰/۵ \times (x_2 - w_2) = [۰/۰۵, ۰/۱۵, -۰/۲۵, -۰/۱۵]$

$\Rightarrow w_2 = w_2 + \Delta w_2 = \boxed{[۰/۹۵, ۰/۸۵, ۰/۲۵, ۰/۱۵]}$

← حال همین مراحل را با نمونه سوم (x_3) می‌رویم:
$x_3 = [۱, ۰, ۰, ۰]$, $w_1 = [۰/۱, ۰/۷, ۰/۸, ۰/۴]$

$w_2 = [۰/۹۵, ۰/۸۵, ۰/۲۵, ۰/۱۵]$

$\Rightarrow$ Competition: $x_3 - w_1 = [۰/۹, -۰/۷, -۰/۸, -۰/۴] \Rightarrow \|x_3 - w_1\| = \sqrt{۲/۱} \simeq \boxed{۱/۴۵}$

$x_3 - w_2 = [۰/۰۵, -۰/۸۵, -۰/۲۵, -۰/۱۵] \Rightarrow \|x_3 - w_2\| = \sqrt{۰/۸۱} = \boxed{۰/۹}$

← پس w_2 برنده شد:

$\Rightarrow \Delta w_2 = ۰/۵(x_3 - w_2) = [۰/۰۲۵, -۰/۴۲۵, -۰/۱۲۵, -۰/۰۷۵]$

$\hookrightarrow \boxed{w_2 = [۰/۹۷۵, ۰/۴۲۵, ۰/۱۲۵, ۰/۰۷۵]}$

← حال همین مراحل را با نمونه چهارم (x_4) می‌رویم:
$x_4 = [۰, ۰, ۰, ۱]$, $w_1 = [۰/۱, ۰/۷, ۰/۸, ۰/۴]$

$w_2 = [۰/۹۷۵, ۰/۴۲۵, ۰/۱۲۵, ۰/۰۷۵]$

$\Rightarrow$ Competition: $x_4 - w_1 = [-۰/۱, -۰/۷, -۰/۸, ۰/۶] \Rightarrow \|x_4 - w_1\| \simeq \boxed{۱/۲۲۵}$

$x_4 - w_2 = [-۰/۹۷۵, -۰/۴۲۵, -۰/۱۲۵, ۰/۹۲۵] \Rightarrow \|x_4 - w_2\| \simeq \boxed{۱/۴۱۵}$

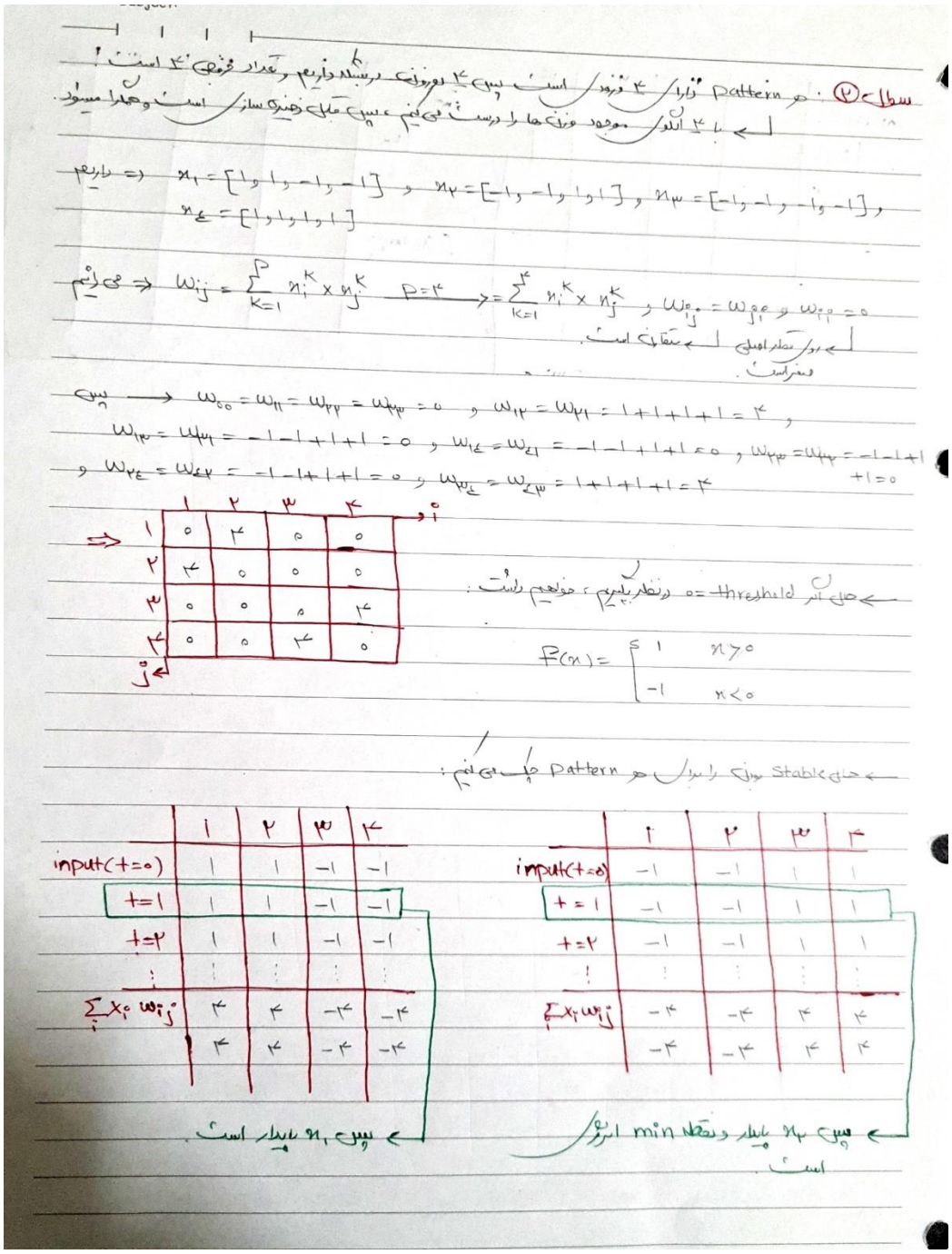
← پس w_1 برنده می‌شود:

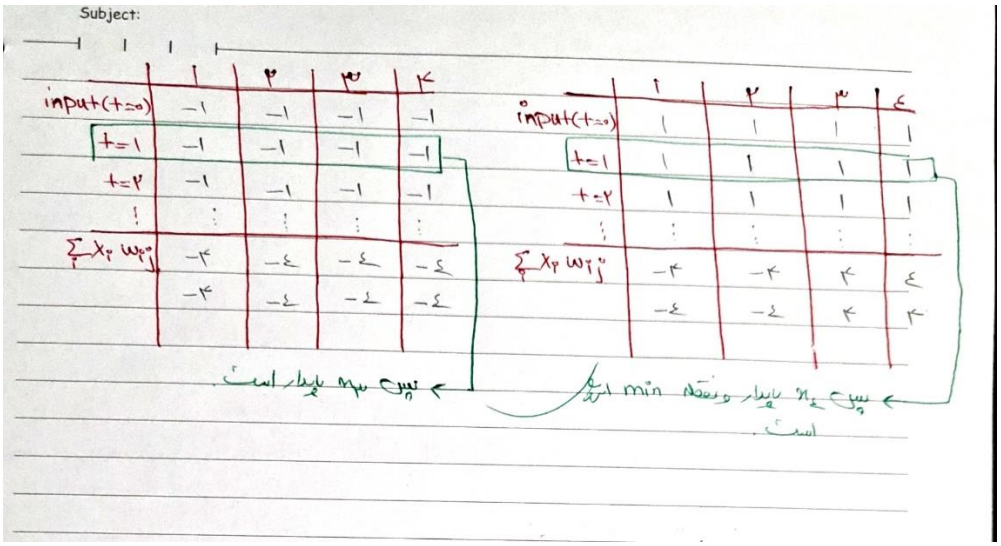
$\Rightarrow \Delta w_1 = ۰/۵(x_4 - w_1) = [-۰/۰۵, -۰/۳۵, -۰/۴, ۰/۳]$

$\hookrightarrow \boxed{w_1 = [۰/۰۵, ۰/۳۵, ۰/۴, ۰/۷]}$

← انقدر این روند ادامه پیدا می‌کند تا همگرا شویم!

سوال 2:


سوال ۲
x1 = [1, 1, -1, -1]
x2 = [-1, -1, 1, 1]
x3 = [-1, -1, -1, -1]
x4 = [1, 1, 1, 1]
threshold = 0
f(x) = 1 if x > 0, -1 if x < 0
input(t=0)
t=1
t=2
Σ xi wij
پس x1 پایدار و نقطه min است.
پس x2 پایدار است.



سوال 3: در این سوال از ما خواسته شده که بدون استفاده از توابع آماده در keras یا مانند آن و تنها با استفاده از کتابخانه numpy مدلی طراحی کنیم تا تابع y = x ^ 2 را تخمین بزند. حال مرحله به مرحله کد زده شده قرار میگیرد و نتیجه نهایی با استفاده از نمودار نشان داده میشود.

مدل طراحی شده شامل سه لایه ورودی، میانی و خروجی است و تابع فعال سازی استفاده شده در لایه میانی RelU میباشد.

- صدا زدن کتابخانه های مورد نیاز:

Libraries

```
[7]   1 import numpy as np
      2 import matplotlib.pyplot as plt
```

- طراحی داده های مورد نیاز برای آموزش و تست: 1000 داده آموزشی و 100 داده تست بصورت رندوم در بازه (3و3-) تشکیل شد.

Generate Data

```
[8]  1 # Generate training data
     2 np.random.seed(42)
     3 X = np.random.uniform(-3, 3, (1000, 1))  # 1000 Input data in range (-3, 3)
     4 y = X ** 2
     5
     6 # Generate test data
     7 X_test = np.random.uniform(-3, 3, (100, 1))
     8 y_test = X_test ** 2
```

- ساختار مدل: تعداد نورون لایه ورودی 1 و تعداد نورون لایه میانی 10 و تعداد نورون لایه خروجی 1 درنظر گرفته شده است:

The architecture of the MLP

```
[9]  1 input_size = 1
     2 hidden_size = 10
     3 output_size = 1
```

- مقدار دهی اولیه متغیرها: بصورت رندوم مقدار دهی اولیه شده اند:

Initialization

```
[10]  1 weights_input_hidden = np.random.rand(input_size, hidden_size)
      2 bias_hidden = np.zeros((1, hidden_size))
      3 weights_hidden_output = np.random.rand(hidden_size, output_size)
      4 bias_output = np.zeros((1, output_size))
```

- آموزش مدل: مدل ساده طراحی شده را طبق داده های آموزشی آموزش دادم و back propagation انجام دادم (کد توضیح هر خط در انتهای آن آمده است):

Training

```python
# Set hyperparameters
learning_rate = 0.01
epochs = 2000

# Training
for epoch in range(epochs):
    hidden_layer_input = np.dot(X, weights_input_hidden) + bias_hidden # Calculate input to hidden layer
    hidden_layer_output = np.maximum(0, hidden_layer_input)  # ReLU activation function
    output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) + bias_output # Calculate input to output layer
    predicted_y = output_layer_input

    # Calculate loss(Mean Squared Error loss)
    loss = np.mean((predicted_y - y) **2 )

    # Backpropagation
    grad_output = 2 * (predicted_y - y) / X.shape[0]

    grad_weights_hidden_output = np.dot(hidden_layer_output.T, grad_output) # Gradient of loss weights in hidden-output layer
    grad_bias_output = np.sum(grad_output, axis=0, keepdims=True) # Gradient of loss bias in output layer

    grad_hidden_output = np.dot(grad_output, weights_hidden_output.T) # Gradient of loss hidden layer output

    grad_hidden_input = grad_hidden_output * (hidden_layer_input > 0) # Gradient of loss hidden layer input

    grad_weights_input_hidden = np.dot(X.T, grad_hidden_input) # Gradient of loss weights in input-hidden layer
    grad_bias_hidden = np.sum(grad_hidden_input, axis=0, keepdims=True) # Gradient of loss bias in hidden layer
```

حال نوبت به آپدیت کردن متغیر ها است:

```python
    # Update weights and biases using gradients and learning rate
    weights_input_hidden -= learning_rate * grad_weights_input_hidden # Update weights in input-hidden layer
    bias_hidden -= learning_rate * grad_bias_hidden # Update bias in hidden layer
    weights_hidden_output -= learning_rate * grad_weights_hidden_output # Update weights in hidden-output layer
    bias_output -= learning_rate * grad_bias_output # Update bias in output layer

    # Print loss every 100 epochs
    if epoch % 100 == 0:
        print(f'Epoch {epoch}, Loss: {loss}')
```

خروجی و ضرر های هر 100 ایپاک به صورت زیر است:

```
Epoch 0, Loss: 9.35762194335822
Epoch 100, Loss: 5.819889076558568
Epoch 200, Loss: 5.250473860684926
Epoch 300, Loss: 4.974190050143854
Epoch 400, Loss: 4.632254303234264
Epoch 500, Loss: 3.9939075739652536
Epoch 600, Loss: 2.639179110443445
Epoch 700, Loss: 1.8532420244163126
Epoch 800, Loss: 1.3745514618692283
Epoch 900, Loss: 1.0586568944137997
Epoch 1000, Loss: 0.8372965578658783
Epoch 1100, Loss: 0.6720655724972386
Epoch 1200, Loss: 0.5428448818258601
Epoch 1300, Loss: 0.4410161512297359
Epoch 1400, Loss: 0.360286983167081
Epoch 1500, Loss: 0.29816952715591055
Epoch 1600, Loss: 0.250419460481235
Epoch 1700, Loss: 0.21334667465335116
Epoch 1800, Loss: 0.18398772464483068
Epoch 1900, Loss: 0.16003137940874895
```

- حال با استفاده از داده های تست، عملیات تست روی مدل ساده طراحی شده انجام میشود و ضرر آن چاپ میشود:

Testing

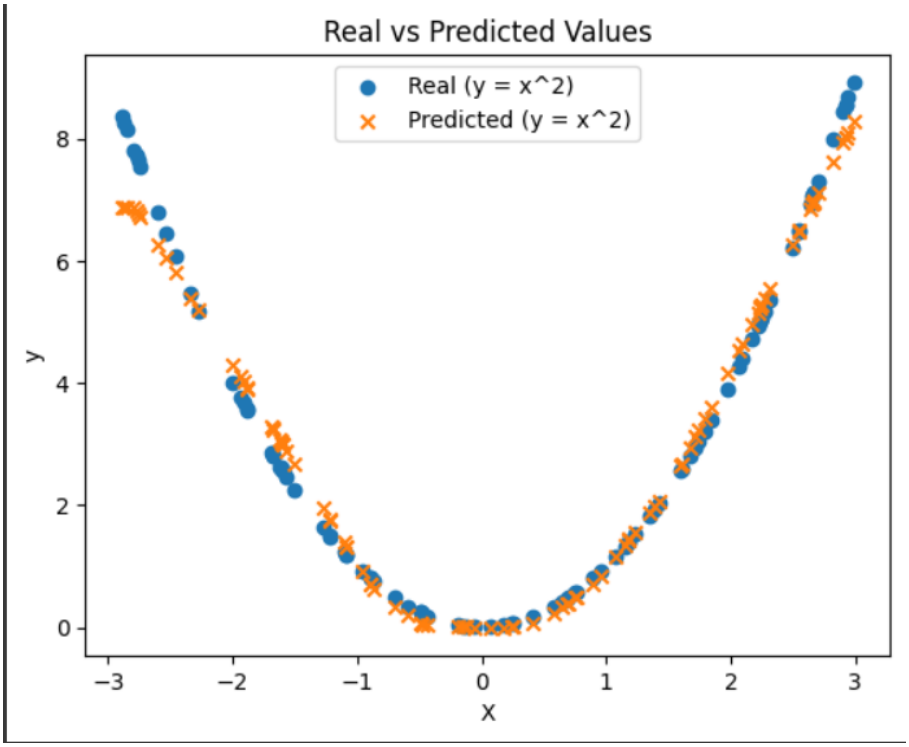
```
hidden_layer_input_test = np.dot(X_test, weights_input_hidden) + bias_hidden # Calculate input to hidden layer for test data
hidden_layer_output_test = np.maximum(0, hidden_layer_input_test) # Apply ReLU activation function for test data
output_layer_input_test = np.dot(hidden_layer_output_test, weights_hidden_output) + bias_output # Calculate input to output layer for test data
predicted_y_test = output_layer_input_test

# Compute loss on the test data(Mean Squared Error loss for test data)
test_loss = np.mean((predicted_y_test - y_test) ** 2)

print(f'Test Loss: {test_loss}')

```

```
Test Loss: 0.1487251580649096
```

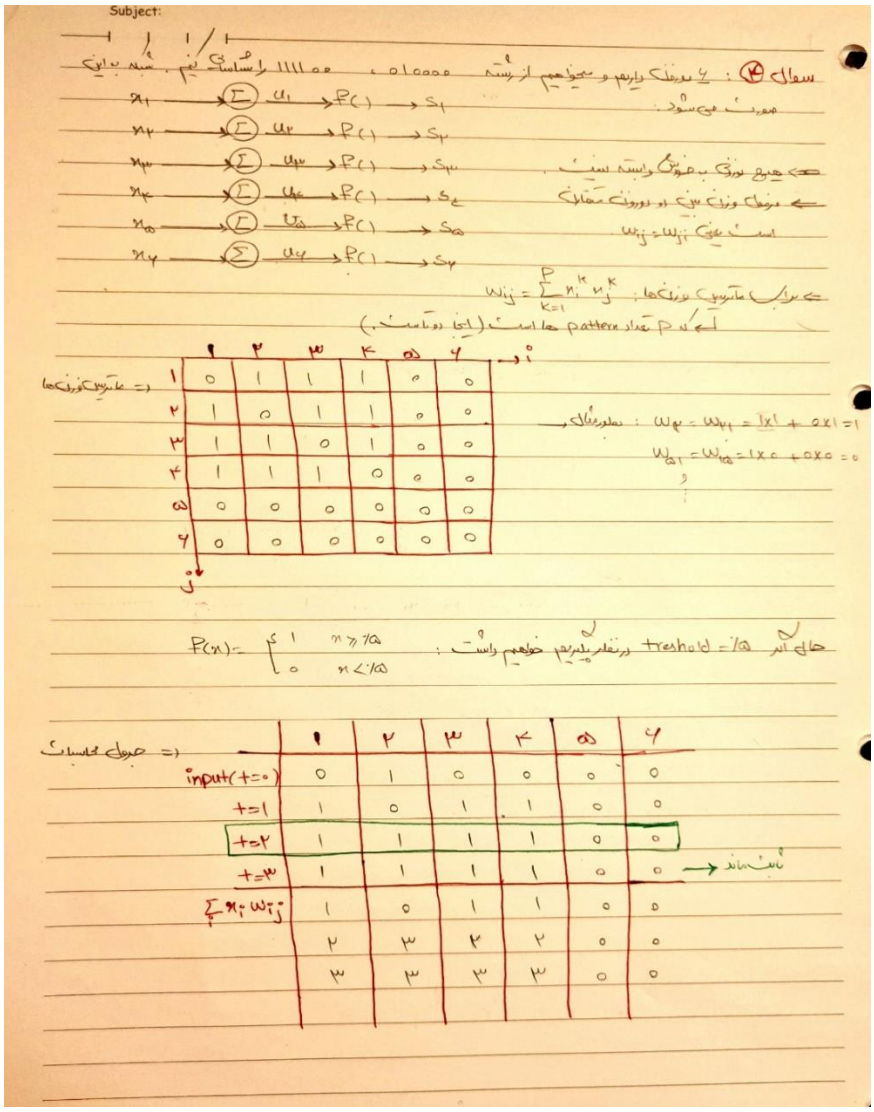
- در نهایت، نمودار تابع واقعی و تابع به دست آمده توسط مدل، رسم شده است:

Ploting the predicted and the real values

```
plt.scatter(X_test, y_test, label='Real (y = x^2)')
plt.scatter(X_test, predicted_y_test, label='Predicted (y = x^2)', marker='x')
plt.title('Real vs Predicted Values')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()
```


Real vs Predicted Values
Real (y = x^2)
Predicted (y = x^2)
y
X
8
6
4
2
0
−3
−2
−1
0
1
2
3

سوال 4:



سوال 5: مسئله TSP یک مسئله بهینه سازی ترکیبی است که در آن یافتن راه حل بهینه با افزایش تعداد شهرها به طور فزاینده ای دشوار می شود.

- شبکه‌های هاپفیلد شبکه‌های عصبی بازگشتی هستند که برای وظایف حافظه انجمنی و مشکلات بهینه‌سازی استفاده می‌شوند.

آنها را می توان برای مدل سازی TSP به عنوان یک مسئله بهینه سازی با تعریف یک تابع انرژی که کل مسافت طی شده را نشان می دهد، استفاده کرد. هر نورون در شبکه مربوط به یک شهر است و اتصالات بین نورون ها نشان دهنده فاصله بین شهرها است. با این حال، آموزش شبکه هاپفیلد برای نمونه‌های بزرگ‌تر TSP به دلیل محدودیت‌های شبکه در مدیریت انفجار ترکیبی و یافتن بهینه جهانی، از نظر محاسباتی فشرده و کمتر مؤثر می‌شود.

- نقشه‌های خودسازماندهی (SOM) مستقیماً برای حل TSP مناسب نیستند زیرا هدف اصلی آنها یادگیری بدون نظارت برای خوشه بندی و کاهش ابعاد است. تطبیق SOMها برای TSP نیاز به اصلاحات اساسی دارد، احتمالاً با تعبیه الگوریتم‌ها یا روش‌های اضافی برای تبدیل مسئله TSP به فرمی مناسب برای SOM(به دلیل ماهیت محدودیت های مشکل (دقیقا یک بار بازدید از هر شهر) چالش برانگیز میباشد.)

- پرسپترون های چندلایه (MLPs) در حالی که همه کاره هستند و قادر به یادگیری نقشه های پیچیده هستند، استفاده مستقیم از آنها برای حل TSP چالش هایی را به دلیل ماهیت گسسته مشکل (انتخاب دنباله ای از شهرها) و نیاز به بازدید از هر شهر دقیقاً یک بار ایجاد می کند. معماری MLP ذاتاً برای مدیریت موثر محدودیت های TSP طراحی نشده است.

شبکه‌های عصبی، همانطور که در بالا توضیح داده شد، ممکن است کارآمدترین یا ساده‌ترین رویکرد برای حل TSP به دلیل ماهیت ترکیبی مشکل نباشند. در عوض، الگوریتم‌های تخصصی مانند الگوریتم‌های ژنتیک، بهینه‌سازی کلنی مورچه‌ها، یا روش‌های برنامه‌نویسی پویا مانند شاخه و کران بیشتر مورد استفاده قرار می‌گیرند و ثابت شده است که نتایج بهتری برای حل TSP دارند.

حال اگر بخواهیم یک شبکه هاپفیلد برای موارد کوچکتر TSP طراحی کنیم، خواهیم داشت:

1. ساختار شبکه:
   a) نورون ها شهرها را نشان می دهند.
   b) اتصال (وزن) بین نورون ها نشان دهنده فاصله بین شهرها است.
   c) یک تابع انرژی را تعریف میکنیم که کل مسافت طی شده را نشان می دهد، با هدف به حداقل رساندن این انرژی.
2. رمز گذاری مشکل: رمزگذاری مشکل TSP در شبکه با مقداردهی اولیه حالات نورون برای نمایش یک توالی خاص از شهرها امکان پذیر است.(وضعیت نورون ها با ترتیب بازدید از شهرها مطابقت دارد.)
3. تابع انرژی:
   a) این تابع کل مسافت طی شده را برای یک دنباله خاص از شهرها نشان می دهد.

b) این تابع باید دنباله های نامعتبر را جریمه کند (به عنوان مثال، بازدید مجدد از یک شهر یا از دست دادن یک شهر).

4. بروز رسانی وزن ها: از به روز رسانی های ناهمزمان یا همزمان برای تنظیم حالت های نورون بر اساس عملکرد انرژی استفاده میکنیم.

5. همگرایی: فرآیند به روز رسانی را تا زمانی که همگرایی یا یک معیار توقف برآورده شود، تکرار میکنیم.

6. خروجی: وضعیت پایدار نهایی شبکه نشان دهنده توالی بهینه شده شهرها (راه حل TSP) است.

همان طور که پیش تر گفته شد، این رویکرد برای نمونه های کوچک TSP کار میکند ولی با افزایش تعداد شهر ها کارآمد نیست و پیچیدگی محاسباتی بالایی دارد.(همچنین ممکن است در مینیمم محلی گیر کند.)

پایان