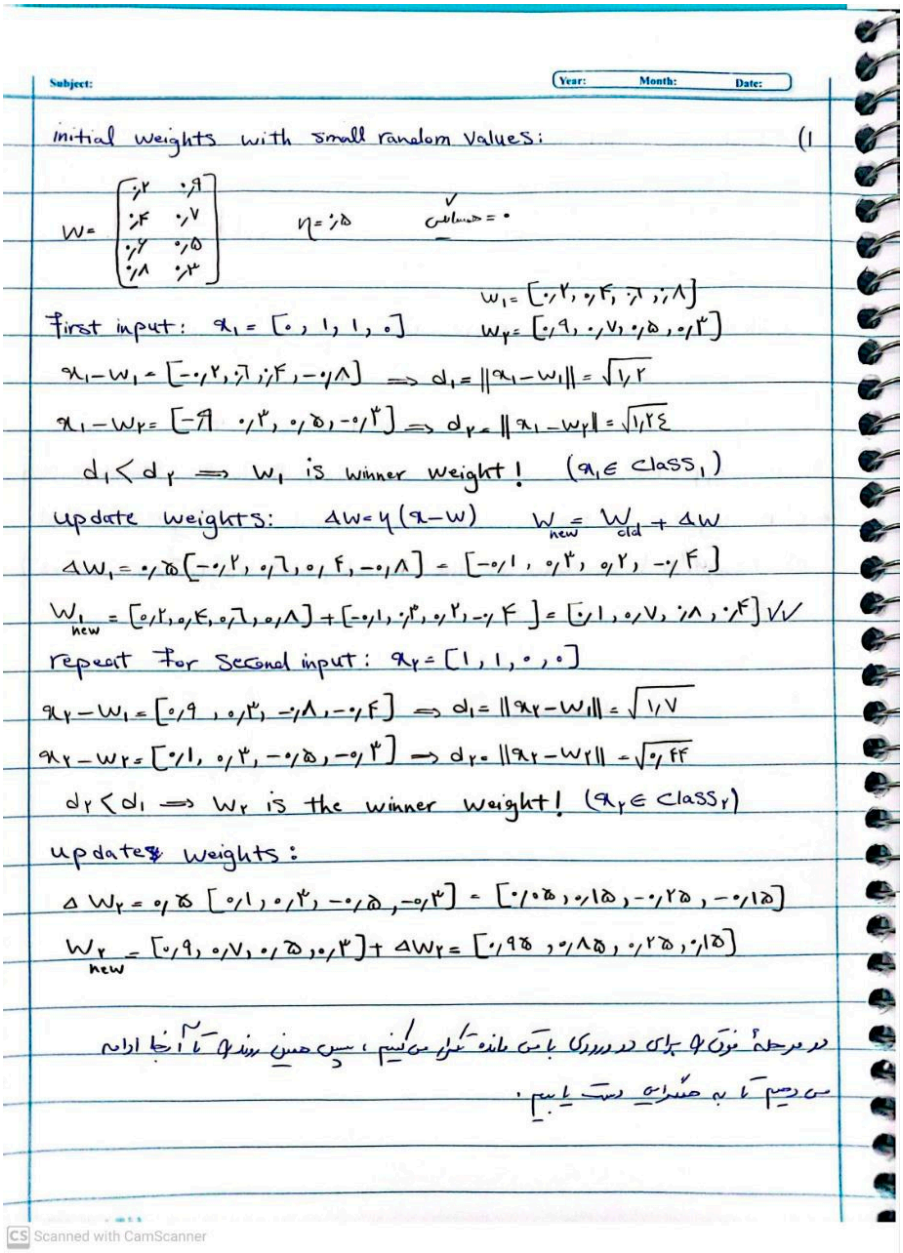
ملیکا محمدی فخار - ۹۹۵۲۲۰۸۶

۱.



Subject: Year: Month: Date:

1) initial weights with small random values:

$$W=\begin{bmatrix} ٪۲ & ٪۹ \\ ٪۴ & ٪۷ \\ ٪۶ & ٪۵ \\ ٪۸ & ٪۳ \end{bmatrix} \qquad \eta = ٪۵ \qquad \text{حساسیت} = 0$$

$w_1 = [٪۲, ٪۴, ٪۶, ٪۸]$

$w_2 = [٪۹, ٪۷, ٪۵, ٪۳]$

first input: $x_1 = [0, 1, 1, 0]$

$x_1 - w_1 = [-٪۲, ٪۶, ٪۴, -٪۸] \Rightarrow d_1 = \|x_1 - w_1\| = \sqrt{1.2}$

$x_1 - w_2 = [-٪۹, ٪۳, ٪۵, -٪۳] \Rightarrow d_2 = \|x_1 - w_2\| = \sqrt{1.24}$

$d_1 < d_2 \Rightarrow w_1$ is winner weight! ($x_1 \in class_1$)

update weights: $\Delta w = \eta(x - w) \qquad w_{new} = w_{old} + \Delta w$

$\Delta w_1 = ٪۵[-٪۲, ٪۶, ٪۴, -٪۸] = [-٪۱, ٪۳, ٪۲, -٪۴]$

$w_{1\,new} = [٪۲, ٪۴, ٪۶, ٪۸] + [-٪۱, ٪۳, ٪۲, -٪۴] = [٪۱, ٪۷, ٪۸, ٪۴]$ ✓✓

repeat for second input: $x_2 = [1, 1, 0, 0]$

$x_2 - w_1 = [٪۹, ٪۳, -٪۸, -٪۴] \Rightarrow d_1 = \|x_2 - w_1\| = \sqrt{1.7}$

$x_2 - w_2 = [٪۱, ٪۳, -٪۵, -٪۳] \Rightarrow d_2 = \|x_2 - w_2\| = \sqrt{0.44}$

$d_2 < d_1 \Rightarrow w_2$ is the winner weight! ($x_2 \in class_2$)

update weights:

$\Delta w_2 = ٪۵ [٪۱, ٪۳, -٪۵, -٪۳] = [٪۰۵, ٪۱۵, -٪۲۵, -٪۱۵]$

$w_{2\,new} = [٪۹, ٪۷, ٪۵, ٪۳] + \Delta w_2 = [٪۹۵, ٪۸۵, ٪۲۵, ٪۱۵]$

در مرحله‌ی نوزدهم برای دو دوره‌ی باقی مانده شبکه را می‌بینیم، سپس همین روند را تا ۲۰ ادامه می‌دهیم تا به همگرایی دست یابیم.

Subject: Year: Month: Date:

۲) Calculate weight matrix : $W_{ij} = \sum_{k=1}^{P} x_i^k x_j^k$

(قطر اصلی درایه‌ها صفر در نظر گرفته می‌شود)

$$W = \begin{bmatrix} 0 & 4 & 0 & 0 \\ 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \\ 0 & 0 & 4 & 0 \end{bmatrix}$$

حالا حدس می‌زنیم که با توجه به ماتریس وزن‌ها، شبکه به ازای هر ورودی استیبل است. چند:

	t_1	t_2	...	$\sum_i x_i w_{ij}$
t_1	1 1 1 1			4 4 4 4
t_2	1 1 1 1			4 4 4 4
⋮	⋮ ⋮ ⋮ ⋮			⋮ ⋮ ⋮ ⋮

	t_1	t_2	...	$\sum_i x_i w_{ij}$
t_1	-1 -1 -1 -1			-4 -4 -4 -4
t_2	-1 -1 -1 -1			-4 -4 -4 -4
⋮	⋮ ⋮ ⋮ ⋮			⋮ ⋮ ⋮ ⋮

	t_1	t_2	...	$\sum_i x_i w_{ij}$
t_1	1 1 -1 -1			4 4 -4 -4
t_2	1 1 -1 -1			4 4 -4 -4
⋮	⋮ ⋮ ⋮ ⋮			⋮ ⋮ ⋮ ⋮

	t_1	t_2	...	$\sum_i x_i w_{ij}$
t_1	1 1 1 1			4 4 4 4
t_2	-1 -1 -1 -1			-4 -4 -4 -4
⋮	⋮ ⋮ ⋮ ⋮			⋮ ⋮ ⋮ ⋮

	t_1	t_2	...	$\sum_i x_i w_{ij}$
t_1	-1 -1 1 1			-4 -4 4 4
t_2	-1 -1 1 1			-4 -4 4 4
⋮	⋮ ⋮ ⋮ ⋮			⋮ ⋮ ⋮ ⋮

همانطور که از جداول مشخص است، تغییری ایجاد نمی‌شود و شبکه به ازای این ورودی‌ها stable است. پس قابل ذخیره‌سازی هستند!

	t_1	t_2	...	$\sum_i x_i w_{ij}$
t_1	1 1 -1 -1			4 4 -4 -4
t_2	1 1 -1 -1			4 4 -4 -4
⋮	⋮ ⋮ ⋮ ⋮			⋮ ⋮ ⋮ ⋮

۳. برای آموزش شبکه نیاز به داده آموزشی داریم، بنابراین ابتدا داده‌ها را به صورت زیر تولید می‌کنیم: (۱۰۰۰ داده آموزشی که اینپوت هرکدام یک عدد رندوم بین ۳ و -۳ است و لیبل آن نیز توان ۲ ورودی است.)

```
np.random.seed(42)
X = np.random.uniform(-3, 3, (1000, 1))
y = X ** 2
```

سپس پارامترهای شبکه را تعیین می‌کنیم. ورودی و خروجی بنا به صورت سوال تنها یک عدد هستند بنابراین برایشان یک نورون در نظر می‌گیریم. با توجه به سادگی مسئله تنها یک لایه میانی با ۱۰ نورون قرار می‌دهیم.

```
input_size = 1
hidden_size = 10
output_size = 1

weights_input_hidden = np.random.rand(input_size, hidden_size)
bias_hidden = np.zeros((1, hidden_size))
weights_hidden_output = np.random.rand(hidden_size, output_size)
bias_output = np.zeros((1, output_size))
```

سپس ماتریس وزن‌ها را به صورت رندوم پر می‌کنیم و بایاس‌ها را ۱ در نظر می‌گیریم.
نرخ آموزش را ۰.۰۱ و تعداد epochها را ۱۵۰۰ در نظر می‌گیریم. حال شبکه را آموزش می‌دهیم.
حلقه آموزش در طول تعداد دورهای مشخص شده از تمام مجموعه داده عبور می‌کند و در هر دوره اقدامات زیر را انجام می‌دهد:

مرحله forward pass:

محاسبه ورودی و خروجی لایه پنهان با استفاده از وزن‌ها و بایاس‌های فعلی.
اعمال تابع فعال‌سازی ReLU بر روی خروجی لایه میانی.

محاسبه خروجی لایه آخر.

محاسبه خطا:
محاسبه خطای میانگین مربعات (mse loss function) بین خروجی پیش‌بینی شده و خروجی واقعی.

مرحله back propagation (محاسبه گرادیان):
محاسبه گرادیان خطا نسبت به خروجی پیش‌بینی شده.
محاسبه گرادیان خطا نسبت به وزن‌ها و بایاس‌های لایه خروجی.
محاسبه گرادیان خطا نسبت به خروجی لایه پنهان.
محاسبه گرادیان خطا نسبت به ورودی لایه پنهان با استفاده از تابع فعال‌سازی ReLU.
محاسبه گرادیان خطا نسبت به وزن‌ها و بایاس‌های لایه پنهان.

به‌روزرسانی وزن و بایاس:
به‌روزرسانی وزن‌ها و بایاس‌های لایه پنهان و خروجی با استفاده از کاهش گرادیان.

ثبت نتایج:
چاپ خطای کنونی هر ۱۰۰ دوره برای نظارت بر پیشرفت آموزش.

```python
# Set hyperparameters
learning_rate = 0.01
epochs = 1500

# Training
for epoch in range(epochs):
    # Forward pass
    hidden_layer_input = np.dot(X, weights_input_hidden) + bias_hidden
    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
    predicted_y = output_layer_input

    # Compute loss (mean squared error)
    loss = np.mean((predicted_y - y)**2)

    # Gradient of the loss with respect to the predicted output
    grad_output = 2 * (predicted_y - y) / X.shape[0]

    # Gradient of the loss with respect to the weights and biases of the output layer
    grad_weights_hidden_output = np.dot(hidden_layer_output.T, grad_output)
    grad_bias_output = np.sum(grad_output, axis=0, keepdims=True)

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

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

    # Gradient of the loss with respect to the weights and biases of the hidden layer
    grad_weights_input_hidden = np.dot(X.T, grad_hidden_input)
    grad_bias_hidden = np.sum(grad_hidden_input, axis=0, keepdims=True)

    # Update weights and biases using gradient descent
    weights_input_hidden -= learning_rate * grad_weights_input_hidden
    bias_hidden -= learning_rate * grad_bias_hidden
    weights_hidden_output -= learning_rate * grad_weights_hidden_output
    bias_output -= learning_rate * grad_bias_output

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

```
Epoch 0, Loss: 11.345666116782384
Epoch 100, Loss: 5.549304169339966
Epoch 200, Loss: 4.849138254867246
Epoch 300, Loss: 3.9944297648338756
Epoch 400, Loss: 2.4877397255886593
Epoch 500, Loss: 1.7284335500759704
Epoch 600, Loss: 1.292705317936548
Epoch 700, Loss: 1.0171361778177175
Epoch 800, Loss: 0.8236577288761623
Epoch 900, Loss: 0.6731008134266172
Epoch 1000, Loss: 0.5530062610231709
Epoch 1100, Loss: 0.4582811036802987
Epoch 1200, Loss: 0.3837469290668148
Epoch 1300, Loss: 0.32341310015454416
Epoch 1400, Loss: 0.27302756457560345
```

در این بخش از کد، داده‌های آزمون تولید شده و سپس خطای مدل روی این داده‌ها محاسبه می‌شود.

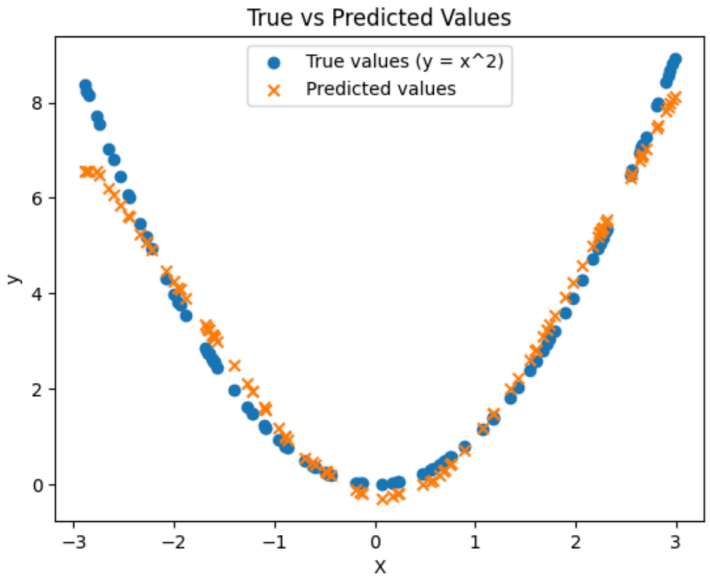
تولید داده‌های آزمون
برای تولید داده‌های آزمون، از یک توزیع یکنواخت در بازه -۳ تا ۳ استفاده شده است. تعداد داده‌های آزمون ۱۰۰ نمونه است.
سپس، ورودی لایه پنهان و خروجی لایه پنهان با استفاده از وزن‌ها و بایاس‌ها محاسبه شده و به تابع فعال‌سازی ReLU داده می‌شود.
با استفاده از این اطلاعات، ورودی لایه خروجی و خروجی پیش‌بینی شده محاسبه می‌شود.
با استفاده از داده‌های آزمون و خروجی پیش‌بینی شده، خطای مدل روی داده‌های آزمون محاسبه می‌شود. در اینجا از تابع خطای میانگین مربعات استفاده شده است.
در نهایت، مقدار خطا بر روی داده‌های آزمون چاپ می‌شود.

```
# Generate test data
X_test = np.random.uniform(-3, 3, (100, 1))
y_test = X_test ** 2

hidden_layer_input_test = np.dot(X_test, weights_input_hidden) + bias_hidden
hidden_layer_output_test = np.maximum(0, hidden_layer_input_test)
output_layer_input_test = np.dot(hidden_layer_output_test, weights_hidden_output) + bias_output
predicted_y_test = output_layer_input_test

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

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

```
Test Loss: 0.23061411488926256
```

در نهایت با استفاده از کتابخانه matplotlib عملکرد مدل بر روی داده‌های آزمون را نمایش می‌دهیم:

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



.۴

Subject: Year: Month: Date:

۴) از آنجا که ورودی‌های شبکه شش بیتی هستند، ۶ نورون در نظر می‌گیریم.
ابتدا ماتریس وزن‌ها W با فرمول $w_{ij} = \sum_{k=1}^{p} x_i^k x_j^k$ محاسبه می‌کنیم. (درایه‌های قطر اصلی ۰ می‌باشد)

111100 → pattern

$$W = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & \\ 1 & 0 & 1 & 1 & 0 & \\ 1 & 1 & 0 & 1 & 0 & \\ 1 & 1 & 1 & 0 & 0 & \\ 0 & 0 & 0 & 0 & 0 & \\ 0 & 0 & 0 & 0 & 0 & \end{pmatrix}$$

حالا بررسی می‌کنیم که آیا سیگنال با وزن‌های محاسبه شده نسبت به pattern داده شده همگرا، stable می‌شود یا خیر:

t_0	۰	۱	۰	۰	۰	۰
t_1	۱	۰	۱	۱	۰	۰
t_2	۱	۱	۱	۱	۰	۰
t_3	۱	۱	۱	۱	۰	۰
⋮	⋮	⋮	⋮	⋮	⋮	⋮
$\sum_i x_i w_{ij}$	۱	۰	۱	۱	۰	۰
	۲	۳	۳	۲	۰	۰
	۳	۳	۳	۳	۰	۰
	⋮	⋮	⋮	⋮	⋮	⋮

حداقل که مشخص است از t_2 به بعد تغییری نداریم و به الگوی مد نظر همگرا شده است.

۵.

از میان سه شبکه عصبی ذکر شده در سوال، شبکه‌ی هاپفیلد برای حل مسئله TSP مناسب‌تر است. دو شبکه دیگر، SOM (کوهونن) و MLP (پرسپترون چندلایه) به دلایلی که در ادامه ذکر خواهد شد، برای این مسئله چندان مناسب نیستند.

شبکه‌ی هاپفیلد

شبکه‌های هاپفیلد نوعی از شبکه‌های عصبی بازگشتی هستند که قابلیت ذخیره و بازیابی اطلاعات گذشته را دارند، که آنها را برای حل مسائل بهینه‌سازی ترکیبیاتی مانند TSP مناسب می‌سازد. در زمینه TSP، یک شبکه هاپفیلد می‌تواند مسئله را به عنوان یک تابع انرژی کد کند، جایی که **انرژی نمایانگر هزینه کل یک تور خاص است.** سپس شبکه وضعیت نورون‌های خود را به‌روزرسانی می‌دهد تا این انرژی را کمینه کند، در نهایت به یک راهحل که نمایانگر یک تور با هزینه بهینه می‌شود، همگرا می‌شود.
از آن جا که شبکه هاپفیلد یک ساختار کاملاً متصل دارد و هر نورون به هر نورون دیگر متصل است برای حل این مسئله مناسب است. زیرا این امکان را به شبکه می‌دهد که روابط بین همه شهرها و فواصل آن‌ها را در نظر بگیرد.

الگوریتم هاپفیلد برای حل مسئله TSP:

۱. کدگذاری مسئله (Problem Representation): هر شهر را به عنوان یک نورون در شبکه هاپفیلد نمایش می‌دهیم. وزن‌های بین نورون‌ها مسافت‌ها بین شهرها را نمایان می‌کنند.

۲. تابع انرژی: یک تابع انرژی تعریف می‌کنیم که از فواصل طولانی جلوگیری می‌کند (با جریمه کردن) و به مسیرهای کوتاهی که از هر شهر دقیقا یک بار عبور می‌کنند مایل می‌شود (با تشویق کردن یا اصطلاحا rewarding)
۳. به‌روزرسانی شبکه: وضعیت هر نورون را بر اساس اصل بهینه‌سازی انرژی محلی، با هدف کاهش کل انرژی شبکه به‌روزرسانی می‌کنیم.

۴. همگرایی: شبکه به‌روزرسانی را تا زمانی که به یک وضعیت پایدار برسد - که نمایانگر یک تور با هزینه کم و رعایت شرط یک بار عبور از هر شهر است - ادامه می‌دهد.

محدودیت‌های SOM و MLP

SOM (کوهونن):
۱. SOM عمدتاً برای وظایف یادگیری بدون نظارت مانند خوشه‌بندی و کاهش ابعاد استفاده می‌شود. این توانایی را برای کنترل مسائل بهینه‌سازی مانند TSP که نیاز به استراتژی‌های جستجو مسیر بهینه دارند، ندارد.
۲. SOM برای حفظ ویژگی‌های توپولوژیکی فضای ورودی در طول نگاشت طراحی شده است. این تمرکز بر حفظ توپولوژی ممکن است با نیازهای بهینه‌سازی TSP که هدف آن کمینه کردن مسیر کوتاهتر است، همخوانی نداشته باشد.
۳. SOM در حالت یادگیری بدون نظارت عمل می‌کنند، جایی که شبکه یاد می‌گیرد داده‌های ورودی را بدون برچسب به صورت منظم سازماندهی دهد. با این حال، TSP یک مسئله نظارت شده است که هدف آن یافتن یک راه‌حل بهینه بر اساس یک معیار خاص است (کمینه کردن مسافت سفر کل). طبیعت بدون نظارت SOM باعث می‌شود که برای وظایفی که نیاز به بهینه‌سازی دارند، کمتر مناسب باشند.

MLP (پرسپترون چندلایه):
۱. هرچند MLP می‌تواند برای مسائل تقریبی استفاده شود، اما با مشکلاتی در مواجهه با مسائل بهینه‌سازی ترکیبیاتی مانند TSP روبه‌رو می‌شود.
۲. TSP شامل یافتن جایگشت بهینه از شهرها برای کمینه کردن مسافت کل است. کنترل بهینه جایگشت‌ها برای معماری‌های استاندارد MLP چالشی است، زیرا این معماری‌ها از ساختارهای خاصی برای درک تقارن و وابستگی‌های ترتیبی در جایگشت‌ها بهرهند نیستند.
۳. MLP از حافظه ذاتی برای تصمیمات گذشته در طول دنباله بازدید از شهرها برخوردار نیستند. TSP که یک مسئله بهینه‌سازی مسیر است، نیازمندی به این دارد که مدل ترتیب بازدید از شهرها را به یاد بسپارد و از اطلاعات توالی به‌طور موثر استفاده کند. MLP ممکن است دچار مشکل شود در حفظ و استفاده مؤثر از این اطلاعات توالی.
۴. TSP شامل اتخاذ یک دنباله از تصمیمات است، که ترتیب بازدید از شهرها را برای بهینه‌سازی هدف خاصی تعیین می‌کند، معمولاً کمینه کردن مسافت سفر کل. MLP به طور ذاتی برای فرآیندهای تصمیم‌گیری متوالی طراحی نشده‌اند، و ممکن است دچار چالش شوند در درک ساختاری که TSP از آن برخوردار است.
۵. عدم وجود مکانیسم صریح جستجو: MLP معمولاً در طول آموزش از روش‌های بهینه‌سازی مبتنی بر گرادیان استفاده می‌کنند، که ممکن است برای طبیعت گسسته و ترکیبی TSP مناسب نباشد. TSP نیاز به استراتژی‌های صریح جستجو برای بررسی ترکیب‌های مختلف ترتیب شهرها دارد، و MLP ممکن است با چالش‌ها در مدیریت چنین وظایف بهینه‌سازی ترکیباتی مواجه شود.

در نتیجه، شبکه‌های هاپفیلد به دلیل قابلیت‌های ذخیره و بازیابی خاطرات و قابلیت کمینه کردن توابع انرژی، به یک راهکار مناسب برای حل مسئله TSP منجر می‌شوند.