Detailed Explanation of the Hopfield Network Code

This Python code implements a Hopfield Network that can store and recall 4 binary vectors. Let's break down the code step-by-step:

1. Import and Define Input Vectors:

Python

```
import numpy as np

# Define the input vectors
vectors = np.array([[1, 1, -1, -1],
                    [1, -1, 1, -1],
                    [-1, 1, -1, 1],
                    [-1, -1, 1, 1]])
```

- We import numpy as np for numerical operations.
- We define a NumPy array named vectors with shape (4, 4).
- This array stores the 4 binary vectors we want the network to learn.
  - Each row represents a vector, and elements are either 1 (-1) representing on (off) states of the neuron.

2. Define Weight Matrix:

Python

```
# Define the weight matrix
weights = np.zeros((4, 4))
for i in range(4):
    for j in range(4):
        if i == j:
            weights[i][j] = 0  # Diagonal elements set to 0
        else:
            weights[i][j] = (vectors[i] @ vectors[j]) / 4
```

- We initialize a weight matrix weights with shape (4, 4) filled with zeros using np.zeros.
- We iterate through all elements of the weight matrix (excluding the diagonal) using nested loops.
  - The diagonal elements (i == j) represent connections between a neuron and itself, which are set to 0.
- For non-diagonal elements, we use the dot product (@) between the corresponding rows

(vectors[i]) and columns (vectors[j]) of the vectors array.

- This calculates the sum of products between corresponding elements of two vectors.
- We divide this sum by 4, which is a common normalization factor in Hopfield networks.
- This process implements the Hebbian learning rule, where connections are strengthened if two neurons have the same value and weakened otherwise.

3. Define Activation Function:

Python

```
# Define the activation function
def activation_function(x):
    if x >= 0:
        return 1
    else:
        return -1
```

- This function defines the activation behavior of each neuron in the network.
- It takes a single input x which represents the weighted sum of inputs to a neuron.
- If x is greater than or equal to 0, the neuron is activated and returns 1 (representing "on").
- Otherwise, the neuron is deactivated and returns -1 (representing "off").

4. Define Hopfield Network Function:

Python

```
# Define the Hopfield Network function
def hopfield_network(x, weights):
    y = np.copy(x)  # Create a copy of input vector
    for i in range(4):
        sum = 0
        for j in range(4):
            sum += weights[i][j] * y[j]
        y[i] = activation_function(sum)
    return y
```

- This function represents the core functionality of the Hopfield network.
- It takes two arguments: x (input vector) and weights (weight matrix).
- First, it creates a copy y of the input vector to avoid modifying the original input.
- Then, it iterates through each neuron (i) in the network.
  - Inside the loop, it initializes a variable sum to 0.
  - It iterates through all other neurons (j) and calculates the weighted sum of inputs to neuron

i.
    - This is done by multiplying the corresponding weight weights[i][j] with the current state y[j] of neuron j and adding it to the sum.
  - Finally, it applies the activation function to the calculated sum and updates the state of neuron i in the y vector.
- The function returns the updated vector y after iterating through all neurons.

5. Test the Hopfield Network Function:

Python

```
# Test the Hopfield Network function
for i in range(4):
    print("Input vector:", vectors[i])
    output = hopfield_network(vectors[i], weights)
    print("Output vector:", output)
    print()
```

- This section tests the hopfield_network function with each of the stored

Continued Explanation - Testing the Network

The last section of the code iterates through each of the stored input vectors (vectors) and tests the hopfield_network function with them.

- **for i in range(4):** - This loop iterates four times, once for each vector stored in the vectors array.
- **print("Input vector:", vectors[i])** - This line prints the current input vector being tested.
- **output = hopfield_network(vectors[i], weights)** - This line calls the hopfield_network function with the current input vector (vectors[i]) and the weight matrix (weights) we calculated earlier. The function updates the input vector based on the network's internal dynamics and returns the resulting output vector. This output vector represents the network's "recollection" of the closest stored pattern based on the provided input.
- **print("Output vector:", output)** - This line prints the output vector returned by the hopfield_network function. Ideally, if the input vector is one of the stored patterns, the output vector should be identical to the original input.
- **print()** - This line prints a blank line for better readability between test cases.

By running this code, you should see the original input vectors printed along with the output vectors after the network has processed them. If the network functions correctly, the output vectors for each test case should be identical to the corresponding input vectors, indicating successful retrieval of stored patterns.

Additional Notes:

- Hopfield networks are effective for content addressable memory, where you can provide a

noisy or incomplete version of a stored pattern and the network attempts to converge to the closest stored pattern.

- This code assumes the network converges after a single iteration. In some cases, multiple iterations might be needed for convergence, which can be implemented using a loop within the hopfield_network function.

I hope this detailed explanation clarifies the code and the functionality of the Hopfield network!

-