

## EXPERIMENT NO. 8

**Aim:** Implement Hebbian learning.

**Theory:**

Hebbian learning is a neurobiologically-inspired learning rule that describes how synaptic connections between neurons can be modified based on their activity. It was first proposed by Donald Hebb in 1949 and is often paraphrased as "cells that fire together, wire together." Hebbian learning is a fundamental concept in neuroscience and has been influential in the development of artificial neural networks and learning algorithms.

### Basic Concepts

- Synaptic Plasticity: Synapses, the connections between neurons, exhibit plasticity, meaning they can change in strength over time. Hebbian learning describes how the strength of synaptic connections can be modified based on the correlation between the activities of pre-synaptic and post-synaptic neurons.
- Correlation: Hebbian learning relies on the principle that if two neurons are active simultaneously, the synaptic connection between them should be strengthened. Conversely, if two neurons are rarely active together, the connection should weaken.

### Hebbian Learning Rule

The Hebbian learning rule can be stated as follows:

"If a synapse repeatedly participates in the firing of the post-synaptic neuron, then the connection between the pre-synaptic and post-synaptic neurons is strengthened."

Mathematically, the Hebbian learning rule can be expressed as:

$$\Delta w_{ij} = \eta \cdot x_i \cdot y_j$$

Where:

$\Delta w_{ij}$  is the change in the synaptic weight between neurons i and j.

$\eta$  is the learning rate, determining the magnitude of weight updates.

$x_i$  is the activity (input) of the pre-synaptic neuron i.

$y_j$  is the activity (output) of the post-synaptic neuron j.

## Mechanisms

Hebbian learning can be implemented using various mechanisms:

- Spike-Timing-Dependent Plasticity (STDP): STDP is a biological phenomenon where the timing of pre-synaptic and post-synaptic spikes determines the direction and magnitude of synaptic changes. If the pre-synaptic spike precedes the post-synaptic spike, the connection strengthens (long-term potentiation, LTP). If the post-synaptic spike precedes the pre-synaptic spike, the connection weakens (long-term depression, LTD).
- Associative Learning: Hebbian learning facilitates associative learning by strengthening the connections between neurons that frequently fire together. This enables the formation of associative memories and learning of correlated patterns in the input data.

## Applications

- Neuroscience: Hebbian learning provides insights into the mechanisms underlying synaptic plasticity and learning in the brain. It helps researchers understand how neural circuits are formed and how memories are encoded and stored.

- **Artificial Neural Networks**: Hebbian learning serves as the basis for learning algorithms in artificial neural networks. It is used to train neural networks for pattern recognition, associative memory, unsupervised learning, and self-organization tasks.
- **Cognitive Science**: Hebbian learning informs theories of learning and memory in cognitive science. It helps explain how humans and animals acquire new skills, form associations, and retrieve information from memory.

Hebbian learning is a fundamental concept in neuroscience and artificial intelligence. It describes how synaptic connections between neurons can be modified based on their activity, leading to the formation of associative memories and learning of correlated patterns. By understanding the principles of Hebbian learning, researchers can develop more biologically plausible models of learning and memory and design more efficient learning algorithms for artificial neural networks.

### **Program Code:**

```
import numpy as np
```

```
# Define the input patterns
```

```
patterns = np.array([[1, 1, 1],  
                    [1, -1, 1],  
                    [-1, 1, 1]])
```

```
# Initialize weights randomly
```

```
num_inputs = patterns.shape[1]
```

```
num_neurons = patterns.shape[0]
```

```
weights = np.random.rand(num_inputs, num_neurons)
```

```
learning_rate = 0.1
```

```
# Perform Hebbian learning
```

```
for pattern in patterns:
```

```
    # Compute the outer product of the pattern with itself
```

```
    outer_product = np.outer(pattern, pattern)
```

```
    # Update the weights using Hebbian learning rule
```

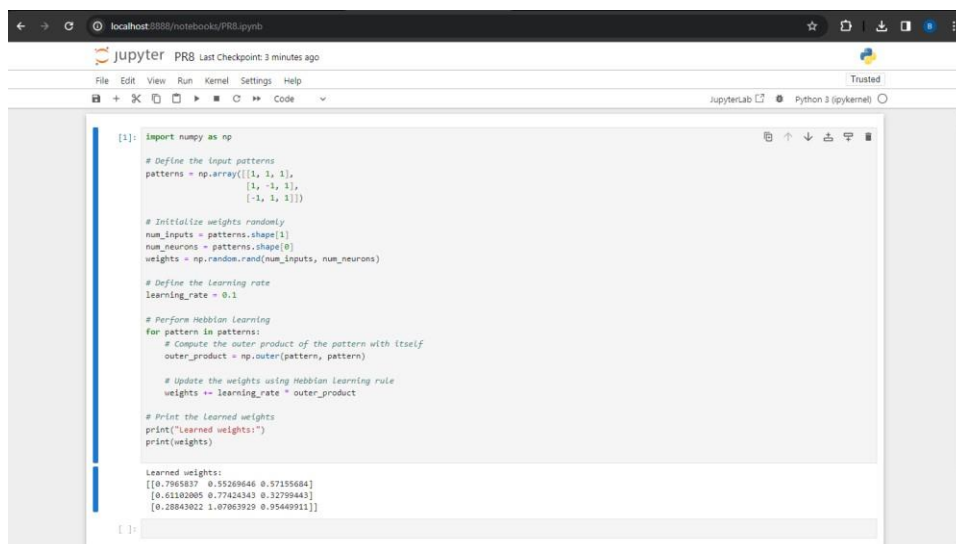
```
    weights += learning_rate * outer_product
```

```
# Print the learned weights
```

```
print("Learned weights:")
```

```
print(weights)
```

## Output:



```
[1]: import numpy as np

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

# Initialize weights randomly
num_inputs = patterns.shape[1]
num_neurons = patterns.shape[0]
weights = np.random.rand(num_inputs, num_neurons)

# Define the learning rate
learning_rate = 0.1

# Perform Hebbian Learning
for pattern in patterns:
    # Compute the outer product of the pattern with itself
    outer_product = np.outer(pattern, pattern)

    # Update the weights using Hebbian learning rule
    weights += learning_rate * outer_product

# Print the learned weights
print("Learned weights:")
print(weights)

Learned weights:
[[0.7965837  0.55269646 0.57155684]
 [0.6102005  0.77424343 0.32799443]
 [0.28843022 1.07063929 0.95449911]]
```

## Conclusion:

By performing this practical we learnt to implement Hebbian learning in python.

