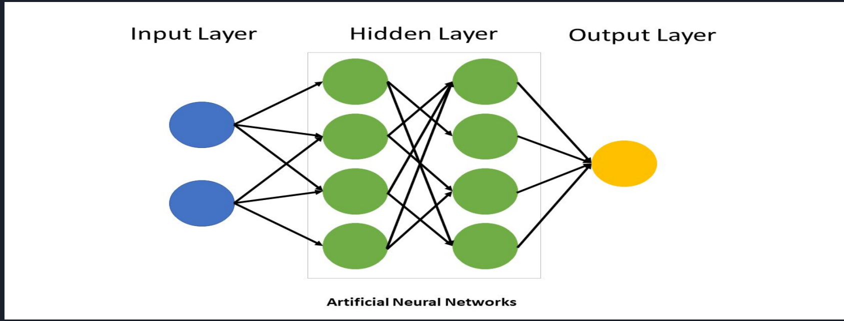
Hebbian Learning Demo

Walkthrough guide on how to create one



Benjamin Bui-Dang | Muskaan Shahzad

Importing Necessities

Imports NumPy for numerical operations and arrays

Imports Matplotlib for creating plots and visualizations

Imports FuncAnimation specifically for creating animations

Sets a random seed (42) to ensure reproducibility of random numbers

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Set random seed for reproducibility
np.random.seed(42)
```

Initialization

Learning in the Brain vs. Machine Learning

Defines a class to encapsulate the two-neuron system

`__init__` is the constructor that initializes the class when an object is created

Sets the initial connection weight between neurons to the provided value (default 0.2)

Sets the learning rate that controls how quickly weights change (default 0.01)

Creates a list to track weight changes over time, starting with the initial weight

Creates an empty list to track correlation between neuron activities

```
class TwoNeuronSystem:
    def __init__(self, learning_rate=0.01, initial_weight=0.2):
        # Initialize connection weight between neurons
        self.weight = initial_weight
        self.learning_rate = learning_rate

        # Store history for visualization
        self.weight_history = [initial_weight]
        self.activity_correlation_history = []
# Explanation:
# Defines a class to encapsulate the two-neuron system __init__ is the construct
# Sets the initial connection weight between neurons to the provided value (defa
# Sets the learning rate that controls how quickly weights change (default 0.01)
# Creates a list to track weight changes over time, starting with the initial we
# Creates an empty list to track correlation between neuron activities

    def simulate(self, num_timesteps=1000, correlation=0.8):
        """
        Simulate the two-neuron system with correlated/uncorrelated inputs.

        Args:
            num_timesteps: Number of simulation steps
            correlation: Correlation between neuron activities (-1 to 1)
        """
```

Section 1

Using Different Methods

Simulate Method: Generates correlated patterns for both neurons which is based on a particular correlation coefficient.

The Plot Results Method: Creates a three section visualization showing weight evolution over a period, activity correlation, and a representation in a visual sense.

The Visualization Method: Updates the connection width reciprocally to the current weight value at each time step.

```
def visualize_dynamic_system(self):
    """Create an animation showing the changing connection strength."""
    fig, ax = plt.subplots(figsize=(8, 6))
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    ax.axis('off')
    ax.set_title('Two-Neuron System with Changing Connection Strength')

    # Create the neurons
    circle1 = plt.Circle((0.3, 0.5), 0.2, color='lightblue', alpha=0.7)
    circle2 = plt.Circle((0.7, 0.5), 0.2, color='lightblue', alpha=0.7)
    ax.add_patch(circle1)
    ax.add_patch(circle2)

    # Add neuron labels
    ax.text(0.3, 0.5, 'N1', ha='center', va='center')
    ax.text(0.7, 0.5, 'N2', ha='center', va='center')

    # Initial weight text
    weight_text = ax.text(0.5, 0.8, f'Weight: {self.weight_history[0]:.2f}',
    step_text = ax.text(0.5, 0.9, f'Step: 0', ha='center')

    # Initial arrow for connection
    arrow = ax.arrow(0.5, 0.5, 0, 0, head_width=0.05,
                     head_length=0.05, fc='black', ec='black',
                     width=self.weight_history[0] * 0.05)
```

Demonstration Function

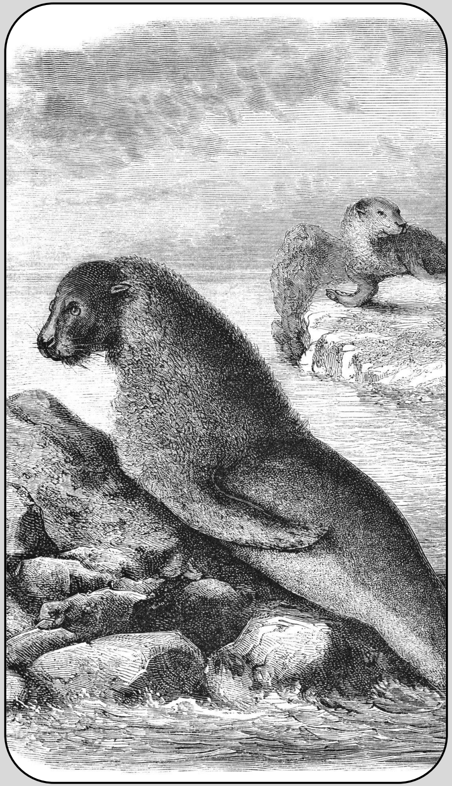
The function creates three two-neuron systems with different activity correlation patterns: positive (0.8), zero (0.0), and negative (-0.8).

Each system demonstrates how connection strength evolves according to Hebbian learning - connections strengthen when neurons fire together (positive correlation), remain relatively unchanged when firing independently (no correlation), and weaken when neurons fire at opposite times (negative correlation).

The function visualizes all results through plots showing weight evolution, correlation over time, and neural representation, with an optional animation showing the dynamic strengthening/weakening process.

```python
# Run simulations with different correlations
def run_demonstration():
    # 1. Positively correlated activity
    print("Simulation 1: Positively Correlated Neurons")
    positive_sys = TwoNeuronSystem(learning_rate=0.01, initial_weight=0.2)
    positive_sys.simulate(num_timesteps=1000, correlation=0.8)
    positive_sys.plot_results()

    # 2. Uncorrelated activity
    print("\nSimulation 2: Uncorrelated Neurons")
    uncorrelated_sys = TwoNeuronSystem(learning_rate=0.01, initial_weight=0.2)
    uncorrelated_sys.simulate(num_timesteps=1000, correlation=0.0)
    uncorrelated_sys.plot_results()

    # 3. Negatively correlated activity
    print("\nSimulation 3: Negatively Correlated Neurons")
    negative_sys = TwoNeuronSystem(learning_rate=0.01, initial_weight=0.2)
    negative_sys.simulate(num_timesteps=1000, correlation=-0.8)
    negative_sys.plot_results()

    return positive_sys, uncorrelated_sys, negative_sys

if __name__ == "__main__":
    positive_sys, uncorrelated_sys, negative_sys = run_demonstration()
```

Conclusion



The simple two neuron system demonstrates the fundamental principle of Hebbian learning in neural networks. Through three distinct scenarios of correlated neural activity, it is revealed through simulations how the strengths of the connections naturally evolve in response to relationships between neurons without external supervision. With this basic implementation along with the straightforward Python code and visualized through the plots and animations, it can capture the essence of neural plasticity that underlies learning, complex biological memory formation, and artificial neural system.