ARTIFICIAL INTELLIGENCE NEURAL NETWORKS

Lab Manual

PART A

1. Simulate a Biological Neuron

```
def biological_neuron(inputs, weights, bias, threshold):  weighted_sum = sum(i * w for i, w in
zip(inputs, weights)) +  bias
 return 1 if weighted_sum > threshold else 0
# Example usage
inputs = [0.5, 0.3, 0.8]
weights = [0.4, 0.6, 0.5]
bias = -0.3
threshold = 0.5
output = biological_neuron(inputs, weights, bias, threshold) print("Neuron output:", output)
```

Expected Output: Neuron output: 1

Note:

- Defines a function called biological_neuron that takes four parameters: inputs, weights, bias, and threshold.
- Calculates the weighted sum of inputs. It pairs each input with its corresponding weight (using zip), multiplies them, and sums them up. Then, it adds the bias to this sum.
- Returns 1 if the weighted_sum exceeds the threshold, indicating the neuron fires; otherwise, it returns 0.
- Example values for inputs, weights, bias, and threshold.
- Calls the function with the example values and stores the output in the variable output.
- Prints the result of the neuron output
- Bias: Start with values like 0 or small values close to 0 (e.g., -0.1, 0.1).
- Threshold: Start with 0.5 or slightly lower based on input expectations.

2. **Basic Artificial Neuron with Sigmoid Activation**

```
import math
def sigmoid(x):
 return 1 / (1 + math.exp(-x))
def artificial_neuron(inputs, weights, bias):
 weighted_sum = sum(i * w for i, w in zip(inputs, weights)) +  bias
 return sigmoid(weighted_sum)
# Example usage
inputs = [0.5, 0.3]
weights = [0.4, 0.6]
bias = -0.2
output = artificial_neuron(inputs, weights, bias) print("Neuron output:", output)
```

Expected Output: Neuron output: 0.549833997

3. **Single Layer Perceptron for XOR Problem**

```
        from sklearn.neural_network import MLPClassifier
        from sklearn.metrics import accuracy_score
# XOR data
X = [[0, 0], [0, 1], [1, 0], [1, 1]]
y = [0, 1, 1, 0]
# Create and train the perceptron
clf = MLPClassifier(hidden_layer_sizes=(2,),
activation='logistic', max_iter=1000)
clf.fit(X, y)
# Predict and evaluate
predictions = clf.predict(X)
print("Predictions:", predictions)
print("Accuracy:", accuracy_score(y, predictions))
```

Expected Output: Predictions: [0 1 1 0], Accuracy:1.0

Note :This is a foundational concept in neural networks, helping tounderstand how neurons learn and make decisions based on input data.This program simulates an artificial neuron using the sigmoid activation function, which processes inputs by multiplying them with corresponding weights, adding a bias, and then applying the sigmoid function to the result. The sigmoid function maps the weighted sum to a value between 0 and 1, representing the neuron's output. You can experiment with different inputs, weights, and biases to see how they affect the output, or modify the program to use other activation functions like ReLU.

4. Multicategory Single Layer Perceptron

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split from sklearn.neural_network
import MLPClassifier from sklearn.metrics import accuracy_score
# Load and split the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y,  test_size=0.3, random_state=42)
# Create and train the perceptron
clf = MLPClassifier(hidden_layer_sizes=(3,), max_iter=1000) clf.fit(X_train, y_train)
# Predict and evaluate
y_pred = clf.predict(X_test)
print("Multicategory Perceptron Accuracy:",
accuracy_score(y_test, y_pred))
```

Expected Output: Multicategory Perceptron Accuracy: ~0.97 (results will vary)

Note: This program implements a multiclass classification task using the Iris dataset with a Multi-Layer Perceptron (MLP) classifier from the scikit-learn library. Here's a breakdown of the code:

1. Import Libraries: The necessary libraries from scikit-learn are imported:
   - load_iris: To load the Iris dataset.
   - train_test_split: To split the dataset into training and testing sets.
   - MLPClassifier: The multi-layer perceptron classifier.
   - accuracy_score: To evaluate the accuracy of the predictions.
2. Load Dataset: The Iris dataset is loaded into the variable iris. The features (input data) are stored in X, and the target labels (species of iris flowers) are stored in y.
3. Split Dataset: The dataset is split into training and testing sets using a 70-30 split (test_size=0.3), with a fixed random state for reproducibility.
4. Create and Train the Classifier: An MLP classifier is created with one hidden layer containing three neurons (hidden_layer_sizes=(3,)) and a maximum of 1000 iterations for training. The classifier is then trained on the training data.
5. Predict and Evaluate: After training, predictions are made on the test set. The accuracy of these predictions is calculated and printed.

5. Delta Learning Rule for MLP

```
import numpy as np
from sklearn.neural_network import MLPClassifier from sklearn.metrics import accuracy_score
# Load and prepare the Iris dataset
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
# Create and train the MLP with Delta learning rule (default) clf =
MLPClassifier(hidden_layer_sizes=(5,), solver='adam',  max_iter=1000)
clf.fit(X, y)
y_pred = clf.predict(X)
print("Delta Learning Rule Accuracy:", accuracy_score(y,  y_pred))
```

Expected Output: Delta Learning Rule Accuracy: ~0.97 (results will vary)

MLPClassifier: The MLPClassifier creates a multi-layer perceptron (MLP), which is a type of neural network that can learn complex patterns in the data.

hidden_layer_sizes=(5,): This parameter specifies the architecture of the network. In this case, it indicates that there is one hidden layer with 5 neurons. The number of hidden layers and neurons can significantly influence how well the network learns from the data.

solver='adam': The 'adam' solver is an optimization algorithm that is widely used for training deep learning models. Adam stands for Adaptive Moment Estimation. It adjusts the learning rate dynamically based on the average of recent gradients (first moment) and the squared gradients

(second moment). This adaptive mechanism is crucial for efficient weight updates, which is a key aspect of delta learning.
fit() Method: When you call clf.fit(X, y), the MLPClassifier begins the training process. It uses the backpropagation algorithm to compute the gradient of the loss function concerning the weights. It calculates the delta (the difference between the predicted and actual output), and then updates the weights accordingly using the Adam optimizer. This is where the delta learning rule comes into play, as it relies on the errors from previous iterations to adjust the model's weights effectively.
accuracy_score: After training, the model predicts the labels for the input data (y_pred = clf.predict(X)). The accuracy of the model is then calculated to evaluate how well the model has learned from the data.

6. Auto-Associative Memory Retrieval

```
import numpy as np
def store_pattern(patterns):
return np.array(patterns)
def retrieve_pattern(stored_patterns, input_pattern): distances = np.linalg.norm(stored_patterns - input_pattern,  axis=1)
return stored_patterns[np.argmin(distances)]
# Example usage
stored_patterns = [[1, 0], [0, 1], [1, 1]]
input_pattern = [0.9, 0.1]
print("Retrieved_pattern:",  retrieve_pattern(np.array(stored_patterns), input_pattern))
```

Expected Output: Retrieved_pattern: [1 0]
Importing NumPy

- importing the NumPy library, which is essential for handling large, multi-dimensional arrays and performing mathematical operations efficiently.

Storing Patterns

- A function named store_pattern is defined to accept a list of patterns.
- This function converts the input list into a NumPy array, allowing for optimized numerical operations.

Retrieving Patterns

- Another function, retrieve_pattern, is defined to find the closest stored pattern based on an input pattern.
- The function calculates the Euclidean distance between the input pattern and each stored pattern.
- It utilizes the np.linalg.norm function to compute the distance for each stored pattern relative to the input.
- The function then identifies the index of the closest pattern using np.argmin, which finds the smallest distance.

- Finally, it returns the stored pattern that corresponds to the smallest distance.

Example Usage

- A list of stored patterns is provided, such as [[1, 0], [0, 1], [1, 1]].
- An input pattern, for instance [0.9, 0.1], is specified to be compared against the stored patterns.
- The retrieve_pattern function is called with the stored patterns and the input pattern, printing the closest match.

Expected Output

- When executed, the program outputs the closest stored pattern to the input pattern.
- For example, if the output is [1, 0], it indicates that this pattern is the closest match to the input pattern [0.9, 0.1].