# Experiment No.9

## Title:Implementation of Single Layer and Multilayer perceptron

## Single Layer Perceptron (SLP)

A **Single Layer Perceptron (SLP)** is the simplest form of a neural network. It consists of an input layer that directly connects to an output layer, with no hidden layers in between. The SLP is mainly used as a linear classifier, which means it can only solve problems that are linearly separable.

---

### Key Components of a Single Layer Perceptron

1. **Inputs (Features)**:
   – The input data that is used for prediction, represented as a vector. For instance, in a binary classification problem, the input might be `[x1, x2, ..., xn]`.
2. **Weights**:
   – Each input is associated with a weight, which signifies how important that feature is for determining the output. The weights are adjusted during the training process.
3. **Bias**:
   – An additional term added to the weighted sum of inputs to shift the decision boundary. It allows the perceptron to classify inputs that are not necessarily centered around the origin.
4. **Weighted Sum**:
   – The perceptron computes the weighted sum of inputs:
   $$z=w_1x_1+w_2x_2+\ldots+w_nx_n+b$$
   where:
   - ( w ) are the weights,
   - ( x ) are the inputs,
   - ( b ) is the bias.
5. **Activation Function**:
   – An activation function decides whether a neuron should be activated or not. For a single layer perceptron, a **step function** or **sign function** is commonly used:
   $$f(z)=\begin{pmatrix}\begin{cases}1 & \text{if } z\geq 0\\ 0 & \text{if } z<0\end{cases}\end{pmatrix}$$
   This function converts the weighted sum into the output (either 0 or 1 in the case of binary classification).

6. **Output**:
   - The perceptron produces a binary output, which can be used for classification tasks (e.g., distinguishing between two classes).

---

## Training a Single Layer Perceptron

Training involves adjusting the weights and bias to minimize the classification error. This is typically done using the **Perceptron Learning Rule**, which updates the weights based on the error between predicted and actual outputs:

$$w_i \leftarrow w_i + \alpha \left( y - \hat{y} \right) x_i$$

where:

- w_i : weight for input x_i
- alpha : learning rate
- y : actual output
- \hat{y} : predicted output

---

## Limitations of Single Layer Perceptron

- **Linearly Separable Data**: A single layer perceptron can only classify data that is linearly separable. This means the classes must be separable by a straight line (in 2D) or a hyperplane (in higher dimensions).
- **No Hidden Layers**: Since there are no hidden layers, the model lacks the complexity to learn more intricate patterns in the data.

---

## Summary

A Single Layer Perceptron is a simple neural network that classifies inputs based on a weighted sum of features. It is an effective model for linearly separable problems but cannot handle more complex, non-linear data. To solve non-linear problems, more advanced networks like **Multi-Layer Perceptrons (MLP)** with hidden layers are needed.

```python
import numpy as np

#Activation function
def sigmoid(x):
  return 1 / (1 + np.exp(-x))

#Another Activation function
def step(x):
  return np.where(x>=0, 1, 0)
```

```
#Single Perceptron with aggrigation and activation
def perceptron(x,w,b):
  return step(np.dot(x,w)+b)

x=np.array([[0,0],[0,1],[1,0],[1,1]])
y=np.array([0,0,0,1])
w=np.array([1,1])
b=-1.5

perceptron(x,w,b)#and gate
```

```
array([0, 0, 0, 1])
```

```
def perceptron_learn(X, y, lr, epochs):
  w = np.zeros(X.shape[1])
  b = 0

  for epoch in range(epochs):
    for i in range(X.shape[0]):
      y_hat = step(np.dot(X[i], w) + b)
      w = w + lr * (y[i] - y_hat) * X[i]
      b = b + lr * (y[i] - y_hat)
  return w, b

perceptron_learn(x,y,1,100)
```

## Multi-Layer Perceptron (MLP)

A **Multi-Layer Perceptron (MLP)** is a type of artificial neural network that consists of multiple layers: an input layer, one or more hidden layers, and an output layer. Unlike a Single Layer Perceptron (SLP), MLPs can solve non-linearly separable problems due to the presence of hidden layers and non-linear activation functions.

---

### Key Components of a Multi-Layer Perceptron

1. **Input Layer**:
   - This layer receives the input features from the dataset. Each feature is represented as an input node in this layer.
   - Example: For a dataset with 3 features, the input layer would have 3 nodes.
2. **Hidden Layers**:
   - MLPs have one or more hidden layers between the input and output layers. Each hidden layer contains neurons that process inputs using weights, biases, and activation functions.
   - The number of neurons and layers can be adjusted to increase the model's capacity to learn complex patterns.

- **Activation Functions**: Non-linear functions like ReLU (Rectified Linear Unit), Sigmoid, or Tanh are applied to the neurons in hidden layers.

$$z^{(l)}=w^{(l)}x^{(l)}+b^{(l)}$$

where:

- ( w^{(l)} ) are the weights for layer ( l )
- ( x^{(l)} ) are the inputs to layer ( l )
- ( b^{(l)} ) is the bias term
- The activation function ( f ) is applied as:

$$a^{(l)}=f\left(z^{(l)}\right)$$

3. **Output Layer**:
   - The output layer produces the final predictions. The number of neurons in the output layer depends on the task:
     - For **binary classification**, there is typically 1 output neuron with a Sigmoid activation function.
     - For **multi-class classification**, the output layer uses the Softmax function with one neuron for each class.
   - **Sigmoid Activation** for binary classification:

$$\sigma(z)=\frac{1}{1+e^{-z}}$$

   - **Softmax Activation** for multi-class classification:

$$\text{Softmax}(z_j)=\frac{e^{z_j}}{\sum_{k=1}^{n} e^{z_k}}$$

---

## How MLP Works

1. **Forward Propagation**:
   - Input data passes through the network layer by layer. Each neuron computes the weighted sum of its inputs, adds a bias, and applies a non-linear activation function to produce an output. This process is repeated for each layer until the final output is produced in the output layer.
2. **Loss Function**:
   - The model uses a loss function to measure how far the predicted outputs are from the actual targets. Common loss functions are:
     - **Binary Cross-Entropy** for binary classification.
     - **Categorical Cross-Entropy** for multi-class classification.

$$L(y,\hat{y})=-\frac{1}{N}\sum_{i=1}^{N}\left(y_i\log(\hat{y}_i)+(1-y_i)\log(1-\hat{y}_i)\right)$$

   where ( y_i ) is the actual label, and ( \hat{y}_i ) is the predicted probability.

3. **Backpropagation**:

- After computing the loss, the MLP uses backpropagation to adjust the weights and biases. The gradients of the loss function with respect to each weight and bias are computed using the chain rule, and the weights are updated using gradient descent:

$$w^{(l)} \leftarrow w^{(l)} - \alpha \frac{\partial L}{\partial w^{(l)}}$$

where alpha is the learning rate.

4. **Optimization**:
   - An optimization algorithm like **Stochastic Gradient Descent (SGD)** or **Adam** is used to minimize the loss function by adjusting the weights iteratively.

---

## Advantages of MLP

- **Non-Linear Problems**: MLPs can learn complex, non-linear relationships between input features and outputs.
- **Multiple Layers**: The presence of hidden layers allows the network to learn hierarchical patterns in data.
- **Universal Approximation**: MLPs with enough neurons and layers can approximate any continuous function.

---

## Limitations of MLP

- **Computational Cost**: Training deep networks with many layers and neurons can be computationally expensive.
- **Risk of Overfitting**: MLPs with too many parameters can overfit to the training data, especially if the dataset is small.
- **Hyperparameter Tuning**: Finding the optimal architecture (number of layers, neurons, learning rate) can be challenging and requires careful tuning.

---

## Summary

A Multi-Layer Perceptron is a powerful neural network that can model complex relationships between inputs and outputs. It works through forward propagation and backpropagation, using non-linear activation functions and optimization algorithms to minimize the loss and improve predictions. While MLPs are versatile, they require careful tuning and computational resources to be effective.

```python
import numpy as np
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# XOR inputs and outputs
```

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 1, 1, 0])

# Initialize the MLPClassifier with a hidden layer of 2 neurons
mlp = MLPClassifier(hidden_layer_sizes=(4,4), activation='relu',
solver='adam', max_iter=5000, random_state=42)

# Train the model on all the data
mlp.fit(X, y)

# Predict on the training set
y_pred = mlp.predict(X)

# Print the results
print("Predictions:", y_pred)
print("Accuracy:", accuracy_score(y, y_pred))
```

```
Predictions: [0 1 1 0]
Accuracy: 1.0
```

## MLP Implementation on a Dataset

```
# Import necessary libraries
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix,ConfusionMatrixDisplay

# Load the Breast Cancer dataset
cancer_data = load_breast_cancer()
X, y = cancer_data.data, cancer_data.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Standardize features by removing the mean and scaling to unit
variance
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create an MLPClassifier model
mlp = MLPClassifier(hidden_layer_sizes=(64, 32),max_iter=1000,
random_state=42)

# Train the model on the training data
mlp.fit(X_train, y_train)
```

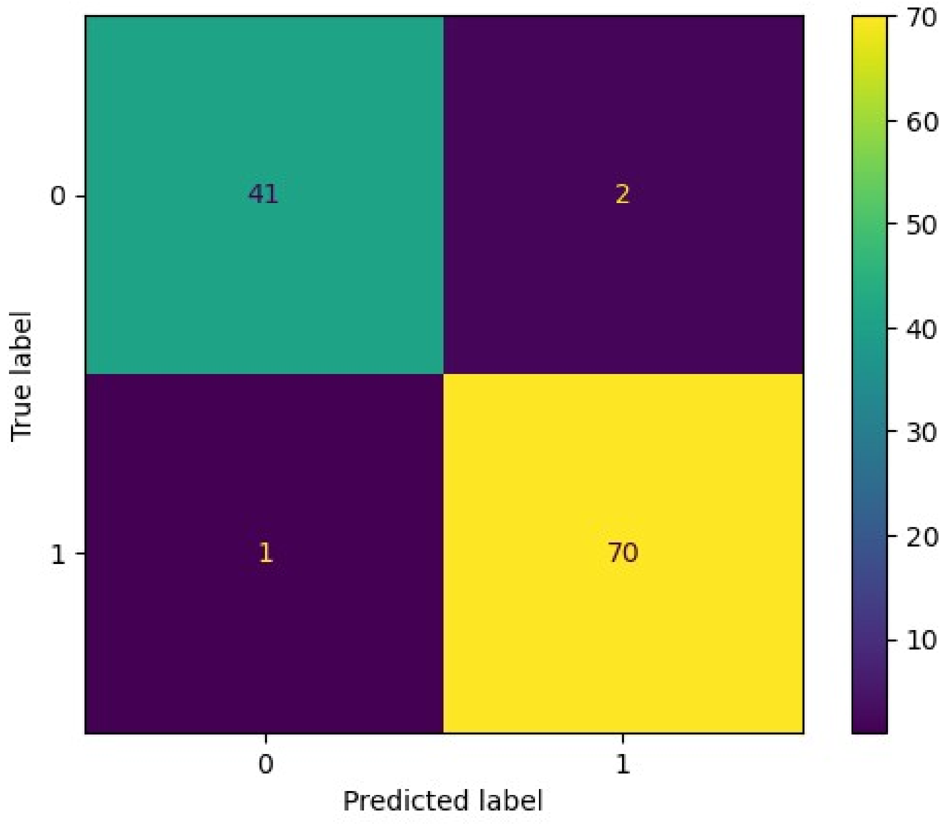
```python
# Make predictions on the test data
y_pred = mlp.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
cm=confusion_matrix(y_test, y_pred)
ConfusionMatrixDisplay(cm).plot()
print(classification_report(y_test, y_pred))
```

```
Accuracy: 0.97
              precision    recall  f1-score   support

           0       0.98      0.95      0.96        43
           1       0.97      0.99      0.98        71

    accuracy                           0.97       114
   macro avg       0.97      0.97      0.97       114
weighted avg       0.97      0.97      0.97       114
```



## Conclusion:

In this experiment, we explored the implementation of both Single Layer Perceptron (SLP) and Multilayer Perceptron (MLP) to solve a classification task, specifically focusing on the XOR problem, which is a classic non-linearly separable problem. The experiment demonstrated the following key conclusions:

1. **Single Layer Perceptron Limitations**: The Single Layer Perceptron, despite being a powerful linear classifier, was unable to solve the XOR problem. This failure highlights the limitation of the SLP in solving problems that require non-linear decision boundaries, as XOR cannot be separated by a single linear hyperplane.
2. **Multilayer Perceptron (MLP) Capability**: The Multilayer Perceptron, equipped with hidden layers and non-linear activation functions like ReLU, successfully solved the XOR problem. This showcases the strength of MLPs in learning non-linear relationships between inputs and outputs by leveraging the depth of the network.
3. **Effect of Hidden Layers**: The presence of hidden layers in MLP allows for hierarchical learning, where the network can learn more complex patterns and transformations. A single hidden layer with a small number of neurons (in our case, two neurons) was sufficient to model the XOR problem.
4. **Training and Convergence**: Increasing the number of training iterations and using an optimizer like Adam ensured that the MLP converged to a solution with high accuracy. The experiment demonstrated that while SLPs are simple and fast, MLPs require more computational resources and training time, but provide significantly better performance for non-linear tasks.
5. **Practical Implications**: This experiment underlines the importance of selecting appropriate model architectures based on the nature of the problem. For linearly separable tasks, SLP may be sufficient, but for more complex problems involving non-linear boundaries, MLPs with hidden layers are essential.

In conclusion, while SLPs are limited to solving linearly separable problems, MLPs with hidden layers significantly expand the range of solvable problems by learning non-linear mappings, making them more suitable for real-world tasks like XOR and beyond.