# Experiment-1

# Perceptron

## 1.Aim

Implementation and Simulation of Perceptron.

## 2.Theory

### Introduction to Perceptrons

The perceptron is the simplest form of artificial neural network, invented by Frank Rosenblatt in 1958. It is a binary classifier that learns to separate data points into two classes using a linear decision boundary. A perceptron computes a weighted sum of its inputs, adds a bias term, and applies a step activation function to produce a binary output.
The basic structure and working of a single-layer perceptron are illustrated in *Fig. 1*.

*"The perceptron is a type of linear classifier, i.e., a classification algorithm that makes its predictions based on a linear predictor function combining a set of weights with the feature vector."*
***- Rosenblatt, 1958***

A single-layer perceptron consists of:

**1. Inputs ($x_1$, $x_2$, ..., $x$□)**

The features or data values that the perceptron receives to make decisions. These are like the raw information fed into the model. These input nodes and their connections to the perceptron are shown in *Fig. 1.*

- Example: For the XOR problem, we have two binary inputs $x_1$ and $x_2$, each can be either 0 or 1

**2. Weights ($w_1$, $w_2$, ..., $w$□)**

These are Importance scores that determine how much each input contributes to the final prediction. A larger weight (positive or negative) means that input has greater influence. The weighted connections between inputs and the summation unit are as seen in *Fig. 1.*

- Each input gets its own weight, acting as a multiplier for that feature
- Weights control the **angle or tilt** of the decision boundary line

**3. Bias (b)**

An adjustable constant added to the weighted sum that shifts the decision boundary's position.

- Bias allows the line to move freely in space to better separate the classes

- Bias controls the **position or shift** of the decision boundary line

**Key Difference Between Weights and Bias:**

- **Weights** determine the slope, how steep the line is and its direction
  **Bias** determines the position, where the line sits in the coordinate space

Together they define a complete line, similar to **y = mx + c**, where weights act like slope 'm' and bias acts like y-intercept 'c'. The bias term $w_o(t) = \theta$

- is as seen in *Fig. 1.*

**4. Net Input (Weighted Sum)**

The perceptron combines all inputs, weights, and bias into a single numerical value:

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_n \cdot x_n + b$$

This can be thought of as calculating a "score", multiply each input by its weight, sum them all together, then add the bias. This score determines the predicted class. The summation operation (Σ

) is as seen in *Fig. 1*.

**5. Activation Function (Step Function)**

The activation function converts the numerical score into a binary decision:

$$\hat{y} = 1 \ if \ z \geq 0$$

$$\hat{y} = 0 \ if \ z < 0$$

- If the score z is positive or zero → predict Class 1
- If the score z is negative → predict Class 0
- This creates a threshold at z = 0 where the perceptron switches between classes
- The resulting decision boundary is always a straight line (linear), which is why perceptrons can only solve linearly separable problems

The step activation function and output of the perceptron are as seen in *Fig. 1*.

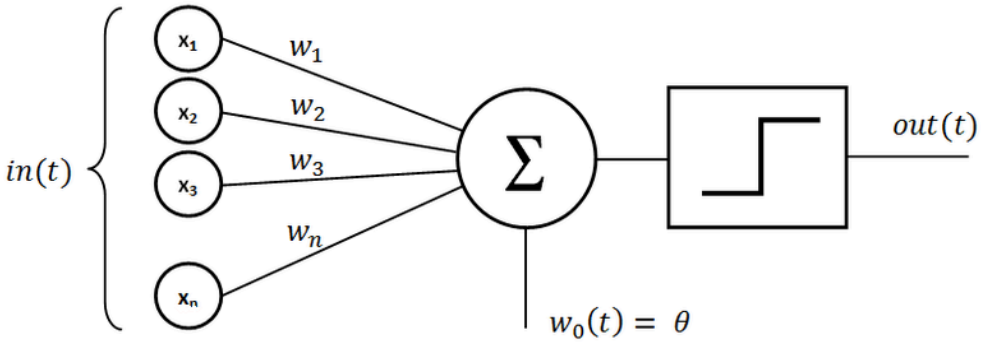



Fig. 1: Architecture of a Single-Layer Perceptron showing inputs, weights, summation unit, bias, and step activation function.

## Perceptron Learning Algorithm

The perceptron learns through a simple trial-and-error process. It makes predictions, checks if they're correct, and adjusts its parameters when it makes mistakes. This is called supervised learning because we provide the correct answers during training.

**The Learning Process:**

**1. Make a Prediction**

The perceptron calculates its weighted sum and makes a prediction:
**$z = w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_n \cdot x_n + b$**

**$\hat{y} = 1$ if $z \geq 0$**
**$\hat{y} = 0$ if $z < 0$**.

**2. Check if the Prediction is Correct**

Compare the predicted output ($\hat{y}$) with the actual correct label ($y$):
**$error = y - \hat{y}$**

$$\text{If } error = 0$$

- : The prediction was correct, no changes needed

$$\text{If } error = +1:$$

- The perceptron predicted 0 but should have predicted 1 (underestimated)

$$\text{If } error = -1$$

- : The perceptron predicted 1 but should have predicted 0 (overestimated)

This prediction process follows the perceptron structure as seen in *Fig. 1*.

**3. Update the Weights (Learn from Mistakes)**

When the perceptron makes an error, it adjusts each weight to reduce that error:

$$w_i = w_i + learning\ rate \times error \times x_i$$

where:

- **Learning rate**: Controls how big the adjustment steps are.
- **error**: Tells us the direction and magnitude to adjust
- **$x_i$**: The input value that contributed to this weight

*Example*: If the perceptron predicted 0 but the answer was 1, the error is +1, so weights connected to active inputs ($x_i = 1$) increase, making the perceptron more likely to predict 1 next time.

**4. Update the Bias (Shift the Boundary)**

The bias is adjusted similarly, but without multiplying by any input:

$$b = b + learning\ rate \times error$$

This shifts the decision boundary to better separate the classes.

**5. Repeat for All Training Examples**

The perceptron goes through every data point in the training set, making predictions and adjusting weights. One complete pass through all data points is called an epoch.

**6. Train for Multiple Epochs**

The process repeats for several epochs. With each epoch:

- The decision boundary moves and rotates
- Classification accuracy typically improves
- The perceptron gets closer to the correct solution (if one exists)

**7. Convergence or Stopping**

Training stops when either:

- **Convergence**: All training examples are classified correctly (only possible for linearly separable data)
- **Maximum epochs reached**: We've trained for a fixed number of epochs

**Important Note**: For linearly separable problems like AND or OR gates, the perceptron will eventually find a perfect solution. However, for non-linearly separable problems like XOR, the perceptron will never converge and will continue making errors indefinitely, which is exactly what we observe in this experiment.

## Linear Separability

A dataset is linearly separable if there exists a straight line (in 2D), plane (in 3D), or hyperplane (in higher dimensions) that can perfectly separate the two classes. Single-layer perceptrons can only learn linearly separable patterns.

**Examples of linearly separable problems:**

AND gate: $(0,0)\rightarrow 0,\ (0,1)\rightarrow 0,\ (1,0)\rightarrow 0,\ (1,1)\rightarrow 1$
OR gate: $(0,0)\rightarrow 0,\ (0,1)\rightarrow 1,\ (1,0)\rightarrow 1,\ (1,1)\rightarrow 1$

**Examples of non-linearly separable problems:**

- XOR gate

### The XOR Problem

XOR (Exclusive OR) is a Boolean logic function that outputs 1 when the inputs are different and 0 when they are the same. The XOR truth table is:

| x1 | x2 | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

When plotted in 2D space, the XOR problem shows a diagonal pattern where:

- Points (0,0) and (1,1) belong to Class 0 (blue)
- Points (0,1) and (1,0) belong to Class 1 (red)

No matter how we adjust w1, w2, and b, we cannot draw a single straight line that achieves this separation. The perceptron will keep updating its weights indefinitely, oscillating between different incorrect solutions, never achieving 100% accuracy.

The solution came with the development of multi-layer perceptrons (MLPs) with hidden layers and non-linear activation functions. A two-layer neural network with at least 2 hidden neurons can solve XOR by creating multiple decision boundaries that, when combined, separate the classes correctly.

### Merits of Perceptrons

- **Simplicity**: Easy to understand and implement
- **Computational Efficiency**: Fast training and prediction for linearly separable problems
- **Foundation**: Forms the basis for understanding more complex neural networks
- **Interpretability**: Decision boundary can be easily visualised and understood

### Demerits of Perceptrons

- **Linear Limitation**: Cannot solve non-linearly separable problems like XOR
- **Binary Classification Only**: Limited to two-class problems in basic form
- **Sensitive to Feature Scaling**: Performance can be affected by input feature scales
- **No Convergence for Non-separable Data**: May oscillate indefinitely without reaching a solution

### Code and Result:

#### Block 1: Import Libraries

**INPUT:**

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

**OUTPUT:**

```
✓ Libraries imported successfully
```

---

### Block 2: Generate Dataset

**INPUT:**

```
data = {
    "x1": [0, 0, 1, 1],
    "x2": [0, 1, 0, 1],
    "label": [0, 1, 1, 1]
}
df = pd.DataFrame(data)
df.head()
```

**OUTPUT:**

| x1 | x2 | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

---

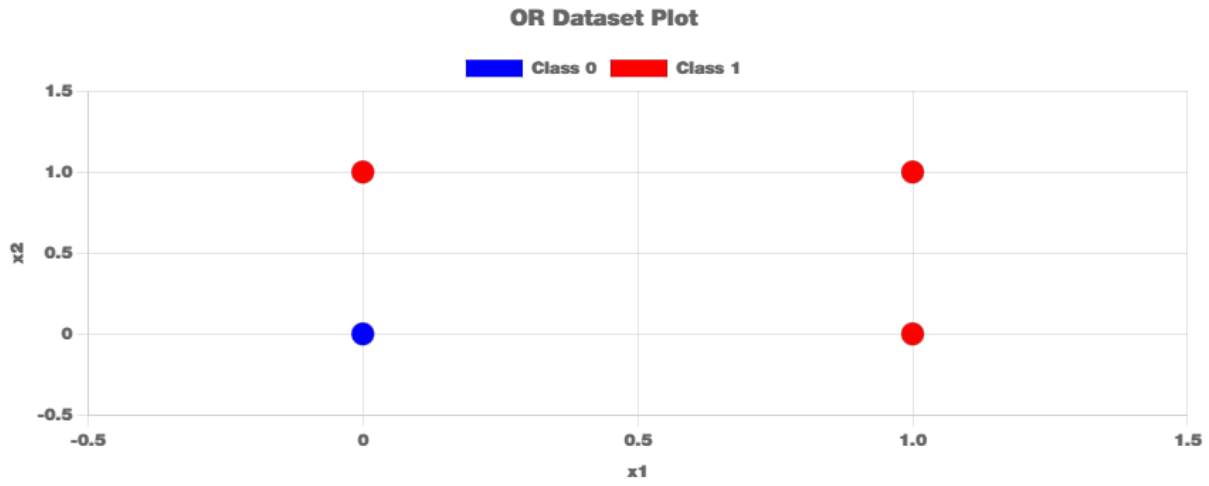
### Block 3: Plot Dataset

**INPUT:**

```
plt.figure(figsize=(6,6))

# Plot class 0
plt.scatter(df[df['label']==0]['x1'],
            df[df['label']==0]['x2'],
            c='blue', s=100, label='Class 0')

# Plot class 1
```

```
plt.scatter(df[df['label']==1]['x1'],
            df[df['label']==1]['x2'],
            c='red', s=100, label='Class 1')

plt.title("OR Dataset Plot")
plt.xlabel("x1")
plt.ylabel("x2")
plt.grid(True)
plt.legend()
plt.show()
```

**OUTPUT:**




---

**Block 4: Initialise Weights**

**INPUT:**

```
np.random.seed(42)
w1 = np.random.uniform(-1, 1)
w2 = np.random.uniform(-1, 1)
b  = np.random.uniform(-1, 1)

learning_rate = 0.1

w1 = -0.4
w2 = 0.2
b = -0.1

print("\nInitial Parameters:")
print(f"w1 = {w1:.3f}, w2 = {w2:.3f}, b = {b:.3f}")
print(f"Learning Rate = {learning_rate}")
```

**OUTPUT:**

Initial Parameters:
w1 = -0.400, w2 = 0.200, b = -0.100
Learning Rate = 0.1

---

**Block 5: Plot Initial Decision Boundary**

**INPUT:**

```
def plot_decision_boundary(w1, w2, b):
    x_vals = np.linspace(-0.5, 1.5, 100)

    if w2 != 0:
        y_vals = -(w1 * x_vals + b) / w2
    else:
        # vertical line if w2 = 0
        y_vals = np.zeros_like(x_vals)

    plt.plot(x_vals, y_vals, label="Decision Boundary", color="green")

plt.figure(figsize=(7,7))

# Plot class 0
plt.scatter(df[df['label']==0]['x1'],
            df[df['label']==0]['x2'],
            c='blue', s=120, label='Class 0')

# Plot class 1
plt.scatter(df[df['label']==1]['x1'],
            df[df['label']==1]['x2'],
            c='red', s=120, label='Class 1')

# Plot boundary
plot_decision_boundary(w1, w2, b)

plt.title("OR Dataset with Initial Random Decision Boundary")
plt.xlabel("x1")
plt.ylabel("x2")
plt.xlim(-0.5, 1.5)
plt.ylim(-0.5, 1.5)
plt.grid(True)
plt.legend()
plt.show()
```

**OUTPUT:**


---

**Block 6: Define Training Function**

**INPUT:**

```
def plot_decision_boundary(df, w1, w2, b, epoch_num):
    plt.figure(figsize=(6,6))

    # Plot points
    plt.scatter(df[df['label']==0]['x1'], df[df['label']==0]['x2'],
                color='blue', s=120, label='Class 0')
    plt.scatter(df[df['label']==1]['x1'], df[df['label']==1]['x2'],
                color='red', s=120, label='Class 1')

    # Boundary
    x_vals = np.linspace(-0.5, 1.5, 100)

    if w2 != 0:
        y_vals = -(w1 * x_vals + b) / w2
        plt.plot(x_vals, y_vals, 'g', label="Decision Boundary")
    else:
        plt.axvline(-b/w1, color='green')

    plt.title(f"Decision Boundary After Epoch {epoch_num}")
    plt.xlim(-0.5, 1.5)
    plt.ylim(-0.5, 1.5)
    plt.grid(True)
    plt.legend()
    plt.show()


# Perceptron Training (Multiple Epochs)
```

```
def perceptron_train(df, w1, w2, b, lr, epochs):

    for epoch in range(1, epochs+1):
        print(f"----EPOCH {epoch}----")

        correct = 0

        for i, row in df.iterrows():
            x1, x2, y = row["x1"], row["x2"], row["label"]

            # Prediction
            linear_output = w1*x1 + w2*x2 + b
            y_pred = 1 if linear_output >= 0 else 0

            # Count accuracy
            if y == y_pred:
                correct += 1

            # Error
            error = y - y_pred

            # Update
            w1 = w1 + lr * error * x1
            w2 = w2 + lr * error * x2
            b  = b  + lr * error

        # Accuracy for this epoch
        accuracy = correct / len(df)
        print(f"Accuracy after Epoch {epoch}: {accuracy*100:.2f}%")
        print(f"Updated Weights -> w1={w1:.3f}, w2={w2:.3f}, b={b:.3f}")

        # Visualise at end of epoch
        plot_decision_boundary(df, w1, w2, b, epoch)

    return w1, w2, b
```

**OUTPUT:**

```
✓ perceptron_train function defined successfully
```

---

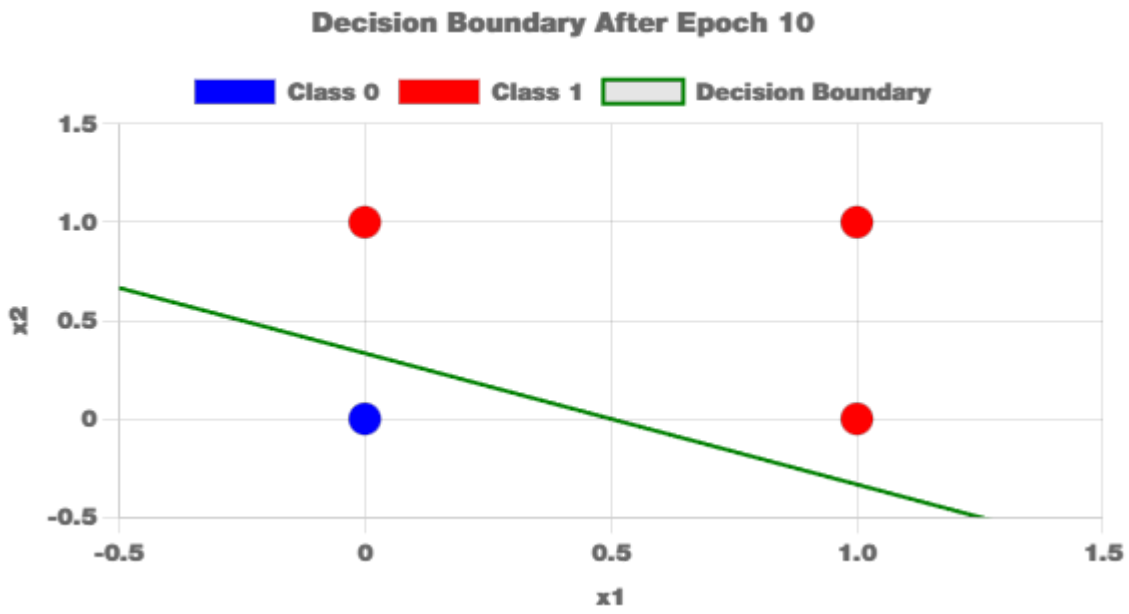
**Block 7: Run Training**

**INPUT:**

```
final_w1, final_w2, final_b = perceptron_train(df, w1, w2, b, learning_rate, epochs=10)

# ---- FINAL EVALUATION AFTER TRAINING ----
def predict(x1, x2, w1, w2, b):
    linear_output = w1*x1 + w2*x2 + b
    return 1 if linear_output >= 0 else 0

correct = 0
total = len(df)

print("\nFinal Predictions After Training:")
for i, row in df.iterrows():
    x1, x2, y = row["x1"], row["x2"], row["label"]
    y_pred = predict(x1, x2, final_w1, final_w2, final_b)
    print(f"Input=({x1}, {x2}) → Predicted={y_pred}, Actual={y}")
    if y_pred == y:
        correct += 1

accuracy = correct / total * 100
print(f"\nFinal Accuracy on Training Data = {accuracy:.2f}%")
```

**OUTPUT:**

```
----EPOCH 10----

Accuracy after Epoch 10: 100.00%

Updated Weights -> w1=0.200, w2=0.300, b=-0.100
```



Final Accuracy:100.00%

| Input (x1,x2) | Predicted | Actual |
|---|---|---|
| (0,0) | 0 | 0 |
| (0,1) | 1 | 1 |
| (1,0) | 1 | 1 |
| (1,1) | 1 | 1 |

---