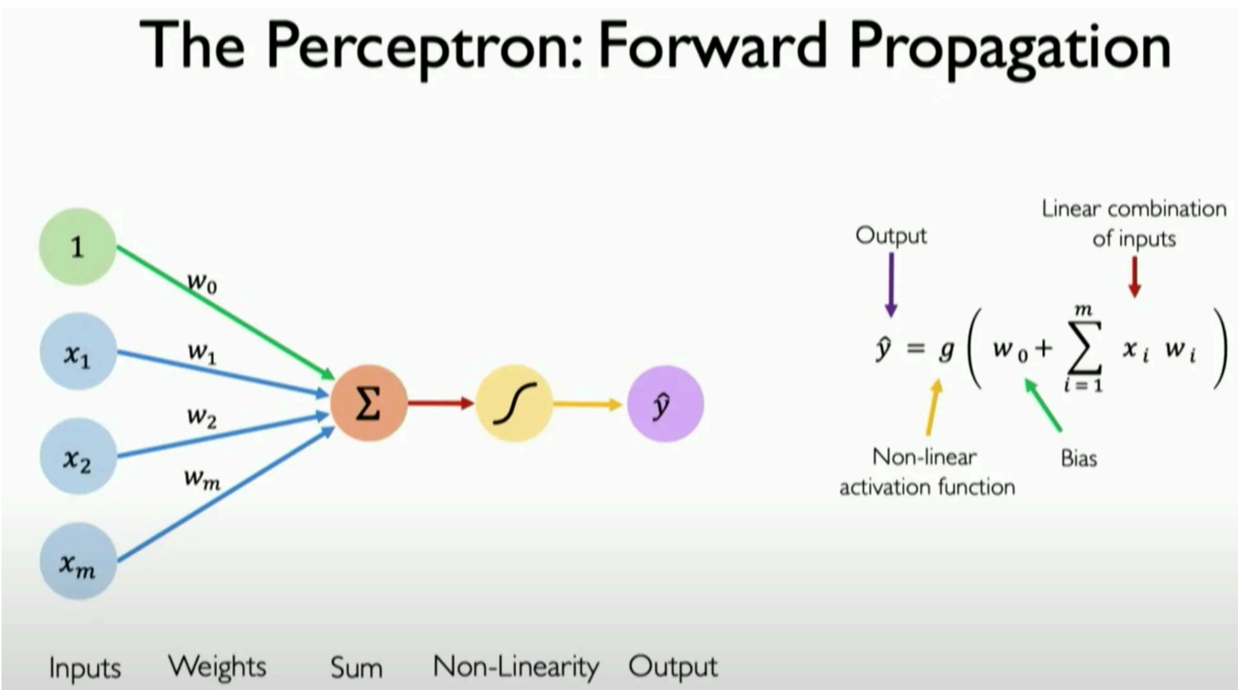
Perceptron

🔷 What is a Perceptron?

A **perceptron** is the **simplest type of artificial neural network**—a single-layer model that makes binary decisions (Yes/No 🟢/🔴). It's the **grandfather of modern deep learning**!

- It's an algorithm used in supervised ML.
- Building block of DL
- It's a mathematical model/Function



🎯 How Does It Work?

1. **Inputs (x_1, x_2, ...)** → Features (e.g., pixel values, temperature).

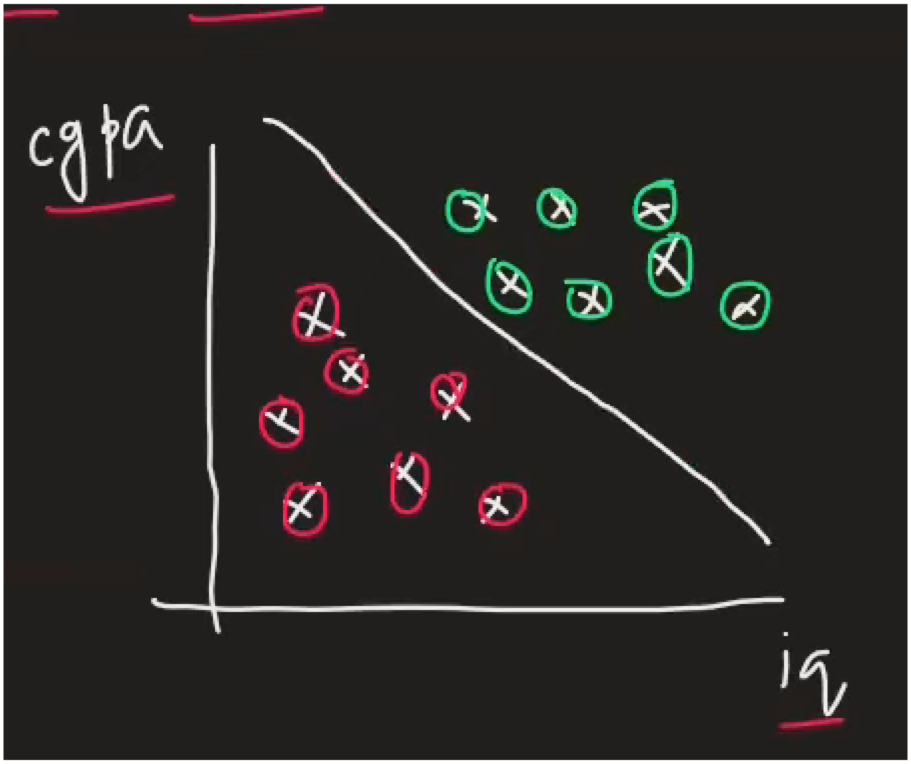
2. **Weights (w_1, w_2, ...)** → Importance of each input.
3. **Bias (b/w_0)** → Adjusts the decision threshold.
4. **Activation Function** → Decides output (e.g., **Step Function**).

$$z = (w_1x_1 + w_2x_2 + w_3x_3 + \ldots) + b$$

$$\text{Output} = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$$

- If **sum ≥ threshold** → Output = **1 (Yes)**
- Else → Output = **0 (No)**

- If weights are more, they play an important role in predicting the output.



- Green is +ve region
- Red is -ve region

> **Perceptron divides region in 2 parts.**

💡 Key Features

✅ **Single-layer** (input + output, no hidden layers).

✅ Learns via **weight updates** (like a simple brain cell 🧠).

✅ Only works for **linearly separable problems** (can't solve XOR ❌).

🆚 Perceptron vs. Modern Neural Networks

Feature	Perceptron	Deep Neural Network (DNN)
Layers	1 (input → output)	Multiple hidden layers
Learning	Basic weight adjustment	Backpropagation + optimization
Use Case	Simple binary classification	Complex tasks (images, NLP)

Feature	Perceptron	Deep Neural Network (DNN)
Limitation	Fails on non-linear data	Handles non-linear patterns

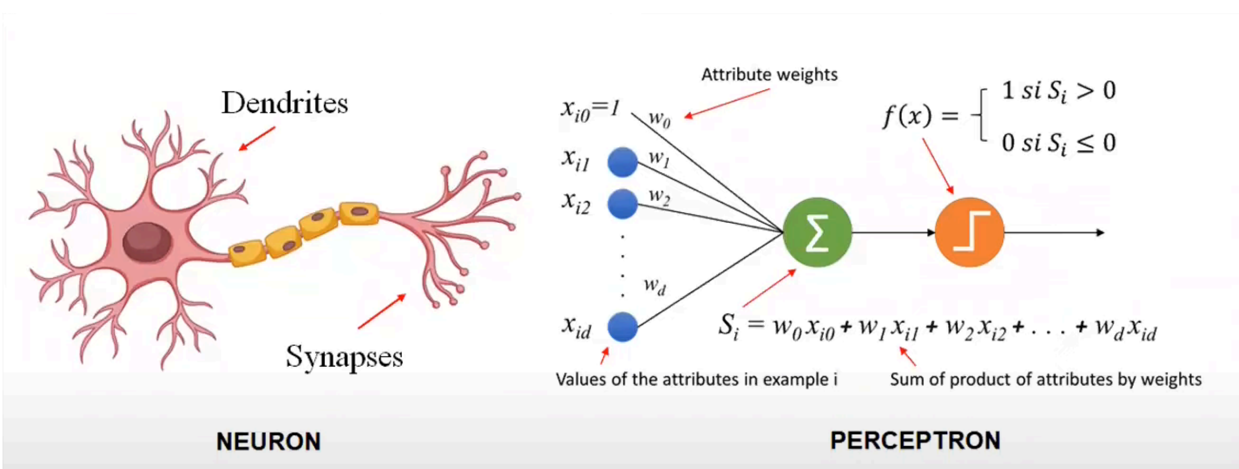
⚙️ Training a Perceptron

1. **Initialize** weights randomly.
2. **For each input**, compute output.
3. **Compare output** with true label.
4. **Update weights** if wrong:

$$w_i = w_i + \alpha \cdot (y - \hat{y}) \cdot x_i$$

- **α (learning rate)**: Controls adjustment speed.
- **y**: True label.
- **ŷ**: Predicted label.

5. **Repeat** until weights converge or for a set number of iterations.
6. Generally, the loop 👆 runs 1000 times



🚀 Evolution: From Perceptron to Deep Learning

- **1958**: Frank Rosenblatt invents perceptron.
- **1969**: Minsky & Papert prove its limits (can't solve XOR).
- **1980s+**: Multi-layer perceptrons (MLPs) + backpropagation fix this!

Limitations of Perceptron

❌ **Cannot solve non-linearly separable problems** (e.g., XOR problem).

❌ **Cannot work on non-linear data**

❌ **Only works for binary classification** (not multi-class).

❌ **Uses a step function**, which does not allow smooth learning.

📌 Summary

- **Perceptron = Simplest neural net** (input → output).
- **Only for linear problems** (e.g., spam vs. not spam).
- **Paved the way for deep learning** 🌟.

Next step? Multi-Layer Perceptrons (MLPs) with hidden layers! 🚀

Python code

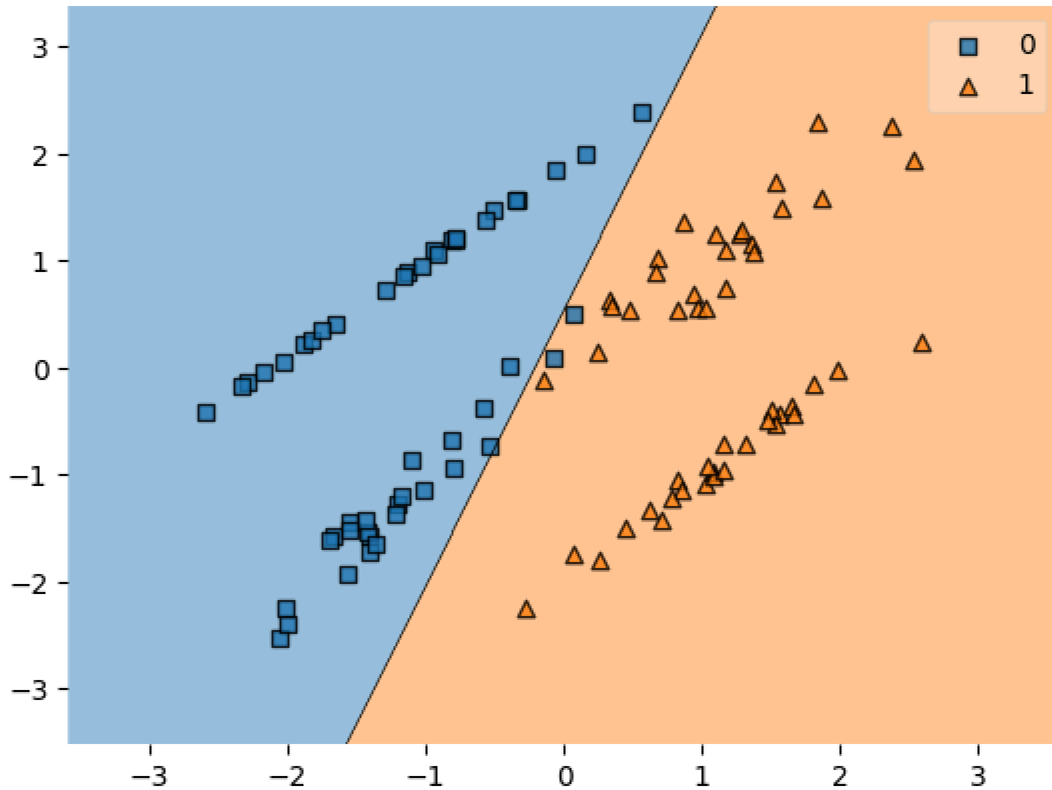
```
import numpy as np
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score

# Create a simple dataset (you can replace it with your own dataset)
```

```
X, y = make_classification(n_samples=100, n_features=2, n_informative=2, n_redundant=0, n_classes=2, random_state=42)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize the Perceptron model
perceptron = Perceptron(max_iter=1000, random_state=42)

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

# Make predictions on the test set
y_pred = perceptron.predict(X_test)

# Calculate and print the accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Print the weights and threshold (bias)
print(f"Weights: {perceptron.coef_}")
print(f"Threshold (Bias): {perceptron.intercept_}")
```

```
Accuracy: 100.00%
Weights: [[ 4.82428394 -1.86776033]]
Threshold (Bias): [1.]
```

- `max_iter=1000` : This is default value

```
from mlxtend.plotting import plot_decision_regions
plot_decision_regions(X, y, clf=perceptron)
```



- **Perceptron** is a **simple linear model**, while other models like **SVM**, **Logistic Regression**, and **Decision Trees** offer more flexibility, can handle complex data, and perform better on non-linear tasks.
- **Perceptron** is easy to implement and interpret but works best only for simple, linearly separable problems.

Key Parameters in `sklearn.linear_model.Perceptron`

Parameter	Description	Default
`eta0`	Learning rate (same as `alpha` in SGD)	`1.0`
`max_iter`	Maximum training iterations (epochs)	`1000`
`random_state`	Seed for reproducibility	`None`
`tol`	Stopping criterion (stops if no improvement)	`1e-3`

Perceptron Trick

- The **Perceptron Trick** is a simple way to **update weights** when the perceptron makes a mistake.
- It's the core learning mechanism behind the perceptron algorithm!

🎯 What is the Perceptron Trick?

If the perceptron **misclassifies** a point:

1. **If prediction = 0 but truth = 1** → **Add** the input vector to weights.
2. **If prediction = 1 but truth = 0** → **Subtract** the input vector from weights.

Update Rule:

$$w_{new} = w_{old} + \alpha(y - \hat{y})x$$

- w = weights
- x = input features
- y = true label (0 or 1)
- $\hat{y}$ = predicted label
- α = learning rate (small step size)

- In the above equation👆, if the point is correctly classified → y=$\hat{y}$
  - Therefore, $w_{new} = w_{old}$
  - i.e. the value of coefficients won't change

Step-by-Step Explanation

1. **Initial Weights (Random)**
    - Starts with a **bad decision boundary** (misclassifies points).
2. **For Each Training Example:**
    - Computes prediction (`y_pred`).
    - **If wrong**: Adjusts weights using:
        - `weights += learning_rate * (y_true - y_pred) * x`
3. **Decision Boundary Moves!**
    - After updates, the boundary shifts to **correctly classify data**.

Loss Function for Perceptron

$$\text{Loss}(w) = -y \cdot (w \cdot x + b)$$

Where:

- w: Weight vector
- x: Input feature vector
- b: Bias term
- y: True label (+1 or -1)
- $w \cdot x$: Dot product between the weights and the input features

How It Works:

1. If the prediction is correct, meaning $y \cdot (w \cdot x + b) > 0$, no loss is incurred, and weights are not updated.
2. If the prediction is incorrect, meaning $y \cdot (w \cdot x + b) \leq 0$, the loss is positive, and the perceptron updates its weights to reduce this loss.

Update Rule:

When the prediction is wrong, we update the weights:

$$w = w + y \cdot x$$