A0597203 AI Business Applications

Introduction to Neural Networks

AI Business Applications

Introduction to Neural Networks

Outcomes

- Fundamentals of neural networks
- Evolution from single Perceptrons to MLPs
- Detailed MLP architecture (input, hidden, and output layers)
- Mathematical representations
- Various activation functions (Sigmoid, ReLU, etc.)
- Backpropagation and training methodologies
- Loss functions and optimization techniques
- Architecture design considerations
- Real-world applications
- Advantages and limitations
- Modern MLP variants and implementations

History of Neural Networks

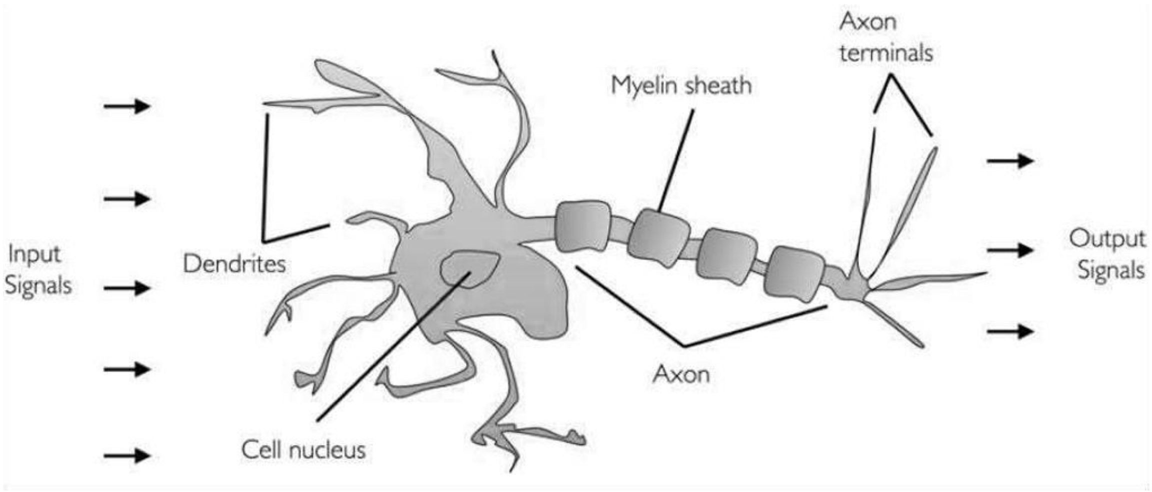
- In 1943, researchers Warren McCullock and published their first concept of simplified brain cell.
- This was called McCullock-Pitts (MCP) neuron.
- They described such a nerve cell as a simple logic gate with binary outputs.
- Multiple signals arrive at the dendrites and are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

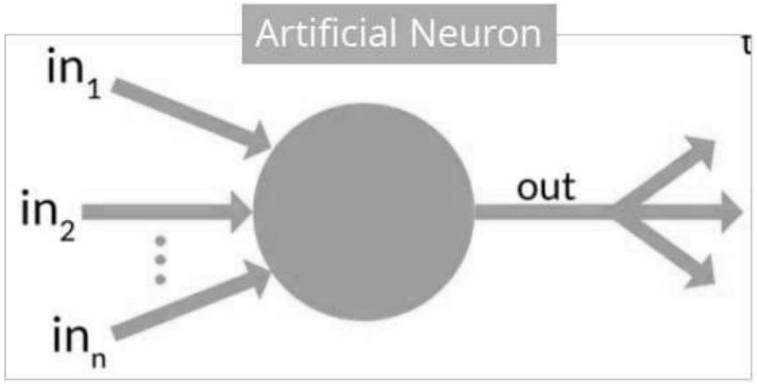
Warren Sturgis McCulloch

(1898 – 1969)

Walter Harry Pitts, Jr.

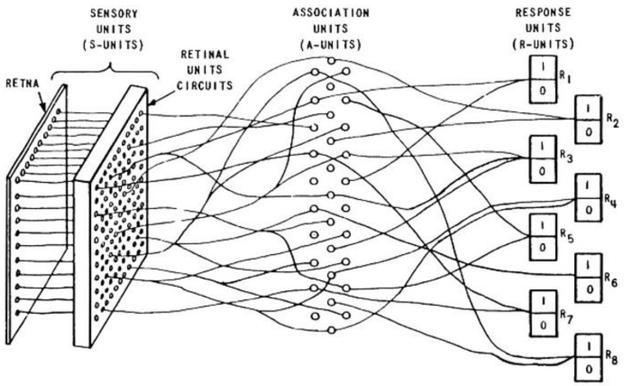
(1923 – 1969)



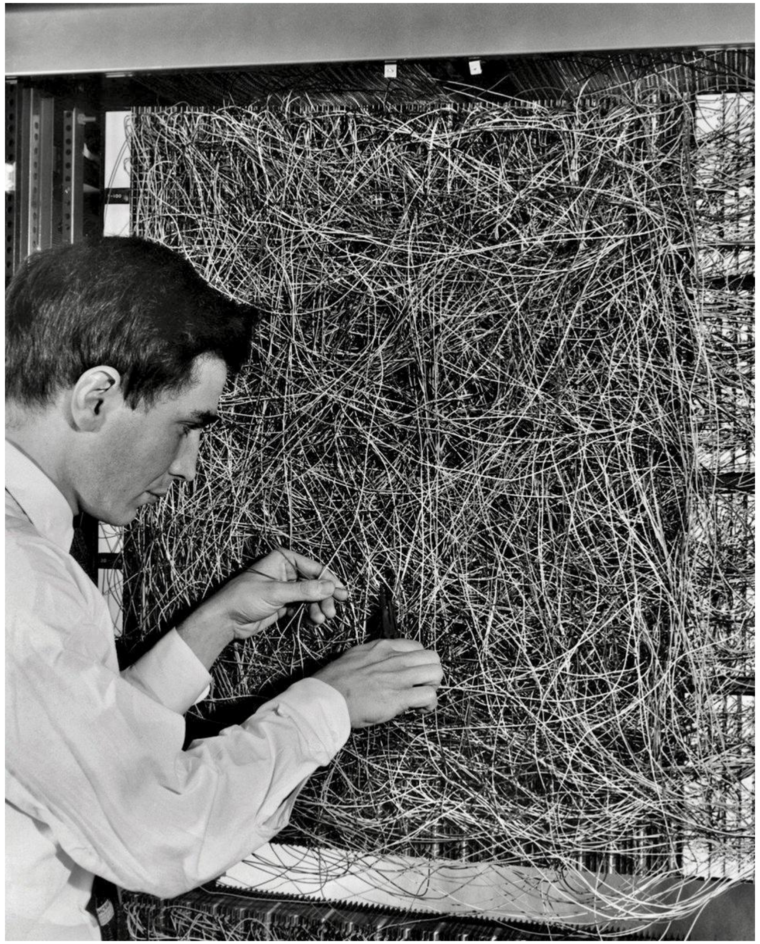


The Perceptron: Building Block of Neural Networks

- In 1953, inspired by McCullock work, Frank Rosenblatt invented the Perceptron.
- The Perceptron is the simplest form of a neural network
- Binary classifier: separates data into two categories
- Models a single neuron with multiple inputs and one output

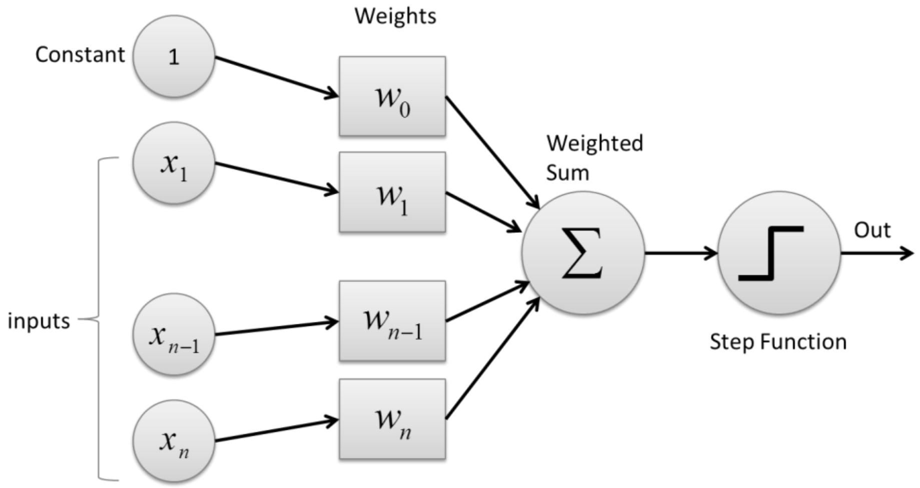


F. Rosenblatt



The Perceptron

- Inputs: $x_1, x_2, \ldots, x_n$
- Weights: $w_1, w_2, \ldots, w_n$
- Bias: b
- Activation function: Step function
- Output: 1 if weighted sum > threshold, 0 otherwise

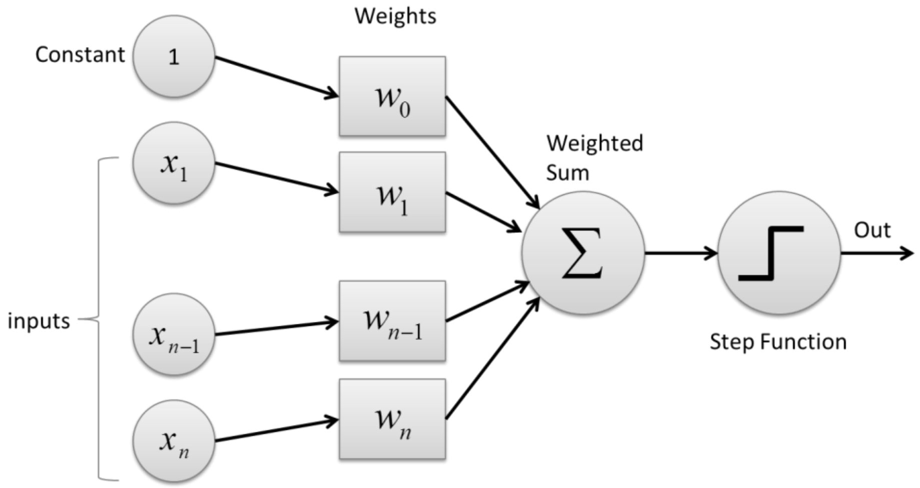


How a Perceptron Works

1. Multiply each input by its corresponding weight
2. Sum all weighted inputs
3. Add the bias term
4. Apply the activation function
5. Output the result

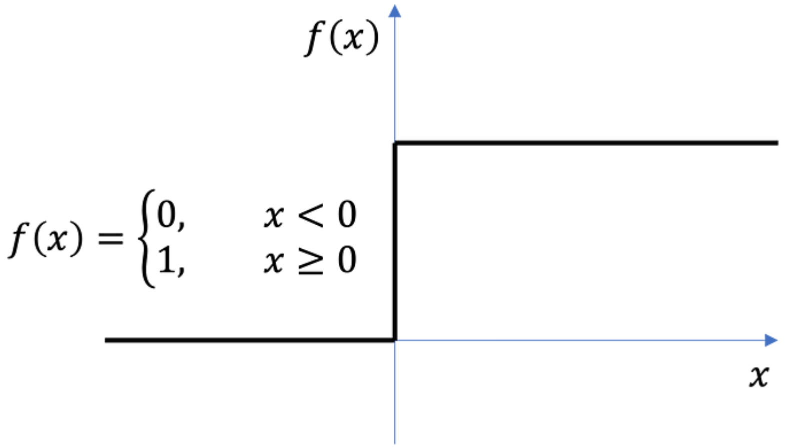
Mathematically:

- $z = w_1x_1 + w_2x_2 + ... + w_nx_n + b$
- output = activation(z)



Perceptron Activation Function

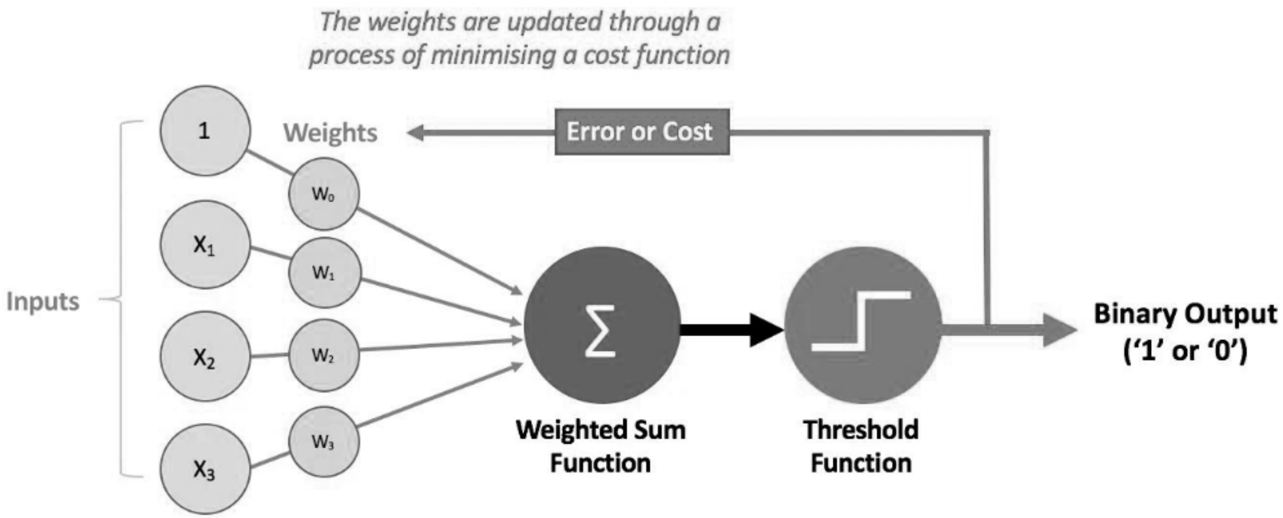
- **Step Function**:
  - Output: 1 if z ≥ 0, 0 if z < 0
  - Used in original perceptrons
  - Not differentiable at 0



How Perceptron Learn (The Cost Function)

For each training example:

1. Calculate predicted output y_pred
2. Calculate error: error = y_true - y_pred
3. Update weights: w_new = w_old + learning_rate * error * x
4. Update bias: b_new = b_old + learning_rate * error



Step-by-Step Hand Calculation for AND Gate

Let's work through the perceptron learning algorithm by hand for the AND gate:

- Training data: X = [[0,0], [0,1], [1,0], [1,1]], y = [0, 0, 0, 1]
- Learning rate (η) = 0.1
- Initial weights (randomly assigned): $w_1 = 0.3$, $w_2 = -0.1$
- Initial bias: $b = 0.2$

First Iteration:

Example 1: (0,0) → 0

- Inputs: $x_1 = 0$, $x_2 = 0$
- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(0) + 0.2 = 0.2$
- Activation: output = 1 (since $z > 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 1 = -1
- Weight updates:
  - $w_1 = w_1 + \eta * \text{error} * x_1 = 0.3 + 0.1 * (-1) * 0 = 0.3$
  - $w_2 = w_2 + \eta * \text{error} * x_2 = -0.1 + 0.1 * (-1) * 0 = -0.1$
  - $b = b + \eta * \text{error} = 0.2 + 0.1 * (-1) = 0.1$

Step-by-Step Hand Calculation for AND Gate

Example 2: (0,1) → 0

- Inputs: $x_1 = 0$, $x_2 = 1$
- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(1) + 0.1 = 0$
- Activation: output = 1 (since $z \geq 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 1 = -1
- Weight updates:
  - $w_1 = w_1 + \eta * error * x_1 = 0.3 + 0.1 * (-1) * 0 = 0.3$
  - $w_2 = w_2 + \eta * error * x_2 = -0.1 + 0.1 * (-1) * 1 = -0.2$
  - $b = b + \eta * error = 0.1 + 0.1 * (-1) = 0$

Example 3: (1,0) → 0

- Inputs: $x_1 = 1$, $x_2 = 0$
- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(1) + (-0.2)(0) + 0 = 0.3$
- Activation: output = 1 (since $z > 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 1 = -1
- Weight updates:
  - $w_1 = w_1 + \eta * error * x_1 = 0.3 + 0.1 * (-1) * 1 = 0.2$
  - $w_2 = w_2 + \eta * error * x_2 = -0.2 + 0.1 * (-1) * 0 = -0.2$
  - $b = b + \eta * error = 0 + 0.1 * (-1) = -0.1$

Step-by-Step Hand Calculation for AND Gate

Example 4: (1,1) → 1

- Inputs: $x_1 = 1$, $x_2 = 1$
- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.2(1) + (-0.2)(1) + (-0.1) = -0.1$
- Activation: output = 0 (since $z < 0$)
- True output: $y = 1$
- Error: error = y - output = 1 - 0 = 1
- Weight updates:
  - $w_1 = w_1 + \eta * \text{error} * x_1 = 0.2 + 0.1 * 1 * 1 = 0.3$
  - $w_2 = w_2 + \eta * \text{error} * x_2 = -0.2 + 0.1 * 1 * 1 = -0.1$
  - $b = b + \eta * \text{error} = -0.1 + 0.1 * 1 = 0$

End of Iteration 1:

- Updated weights: $w_1 = 0.3$, $w_2 = -0.1$
- Updated bias: $b = 0$

Second Iteration

Example 1: (0,0) → 0

- Inputs: $x_1 = 0$, $x_2 = 0$
- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(0) + 0 = 0$
- Activation: output = 1 (since $z \geq 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 1 = -1
- Weight updates:
    - $w_1 = w_1 + \eta * \text{error} * x_1 = 0.3 + 0.1 * (-1) * 0 = 0.3$
    - $w_2 = w_2 + \eta * \text{error} * x_2 = -0.1 + 0.1 * (-1) * 0 = -0.1$
    - $b = b + \eta * \text{error} = 0 + 0.1 * (-1) = -0.1$

Example 2: (0,1) → 0

- Inputs: $x_1 = 0$, $x_2 = 1$
- Weighted sum: $z = w_1x_1 + w_2x_2 + b = 0.3(0) + (-0.1)(1) + (-0.1) = -0.2$
- Activation: output = 0 (since $z < 0$)
- True output: $y = 0$
- Error: error = y - output = 0 - 0 = 0
- Weight updates (no change as error = 0):
    - $w_1 = 0.3$
    - $w_2 = -0.1$
    - $b = -0.1$
- **After several iterations**, the perceptron will converge to weights that correctly classify all AND gate examples.

Python Implementation Perceptron from Scratch

```python
from sklearn.linear_model import Perceptron
import numpy as np

   # Training data for AND gate
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])

   # Initialize and train Perceptron
model = Perceptron(max_iter=100, eta0=0.1, random_state=42)
model.fit(X, y)
# Results
print("Weights:", model.coef_)
print("Bias:", model.intercept_)
print("Predictions:", model.predict(X))
```

```
Weights: [[0.2 0.2]]
Bias: [-0.2]
Predictions: [0 0 0 1]
```

The code shows a scikit-learn Perceptron implementation for the AND gate problem.

The code:

1. Imports NumPy, scikit-learn's Perceptron, and matplotlib
2. Sets up the training data for the AND gate
3. Initializes a Perceptron with 100 max iterations and a random seed of 42
4. Trains the perceptron on the AND gate data
5. Prints the learned weights, bias, and predictions

The output shows:

- **Weights: [[0.2 0.2]]** - The perceptron learned to assign a weight of 0.2 to both inputs
- **Bias: [-0.2]** - The bias is -0.2
- **Predictions: [0 0 0 1]** - The perceptron correctly classified all four examples of the AND gate

With these weights and bias, the decision function is: 0.2×(input1) + 0.2×(input2) – 0.2

For the four input combinations:

- [0,0]: $0.2\times0 + 0.2\times0 - 0.2 = -0.2 < 0 \rightarrow$ output 0
- [0,1]: $0.2\times0 + 0.2\times1 - 0.2 = 0 \rightarrow$ output 0
- [1,0]: $0.2\times1 + 0.2\times0 - 0.2 = 0 \rightarrow$ output 0
- [1,1]: $0.2\times1 + 0.2\times1 - 0.2 = 0.2 > 0 \rightarrow$ output 1

This perceptron implements the AND gate logic.

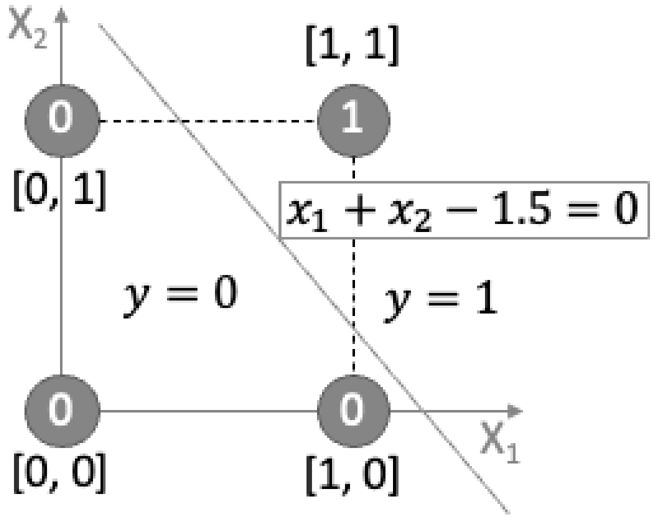
The decision boundary is the line $2x_1 + 2x_2 - 0.2 = 0$, which separates the point (1,1) from the other three points.

Open in Colab

Decision Boundary

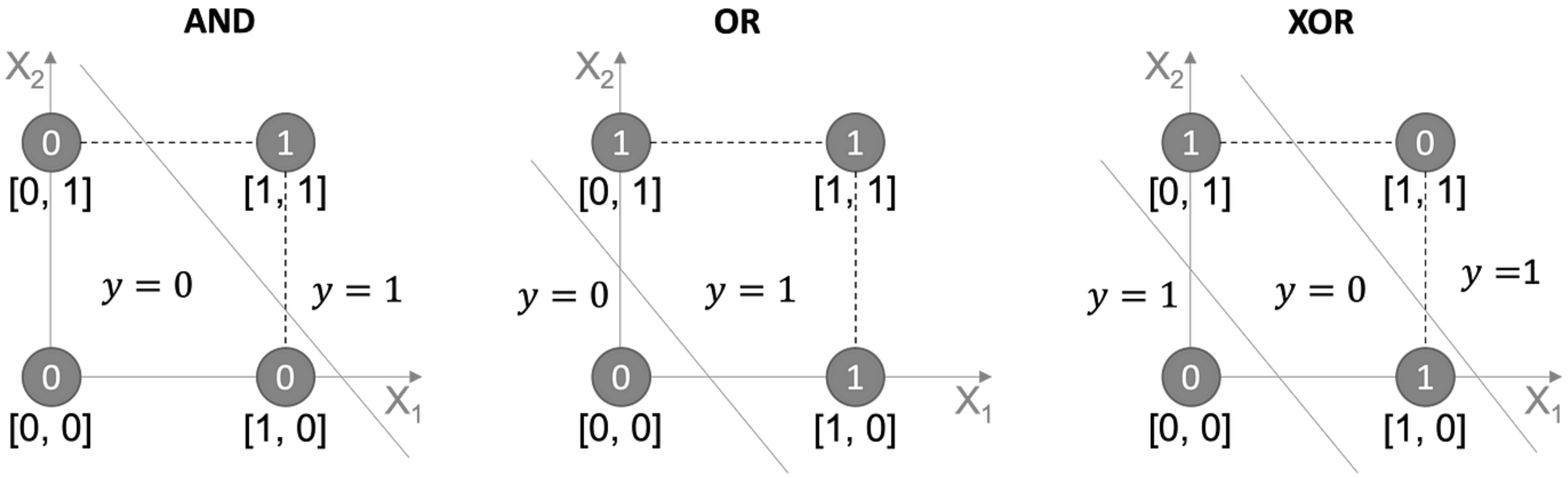
- The perceptron learns a decision boundary: $w_1x_1 + w_2x_2 + b = 0$
- Points above the line are classified as 1
- Points below the line are classified as 0
- For AND gate, only the point (1,1) should be above the line

X_1	X_2	y
0	0	0
0	1	0
1	0	0
1	1	1



Limitations of Simple Perceptron

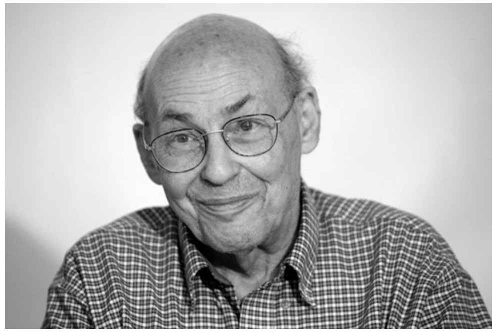
- Can only learn linearly separable patterns
- Cannot solve XOR problem (need multiple layers)
- No probabilistic output
- Simple update rule isn't suitable for complex problems



Introduction to Neural Networks

Multi-Layer Neural Networks

The Multi-Layer Percecptron (MLP)



Limitations of the Perceptron

While useful for linearly separable problems, the single perceptron cannot solve complex problems like XOR classification, as demonstrated by Minsky and Papert in their 1969 book "Perceptrons.“

The Multi-Layer Perceptron

The Multi-Layer Perceptron addresses the limitations of the single perceptron by introducing:

- Multiple layers of neurons
- Non-linear activation functions
- More sophisticated learning algorithms

Structure of an MLP

Definition: An MLP is a class of feedforward artificial neural network that consists of at least three layers of nodes: **input**, **hidden**, and **output** layers.

Key Feature: Each neuron in one layer is connected to every neuron in the next layer (fully connected).

1. Input Layer

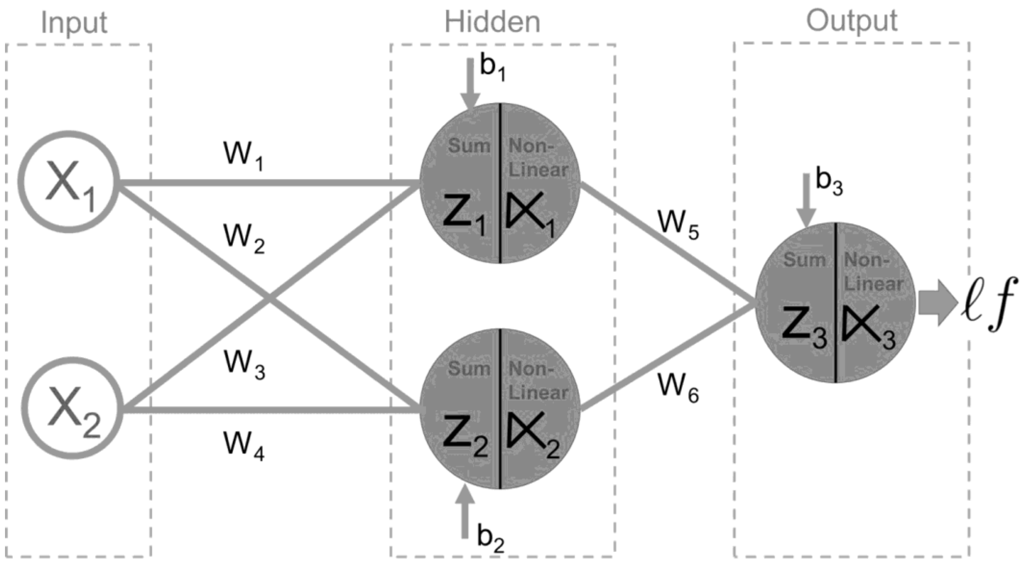
- Receives the raw input features
- One neuron per input feature
- No computation occurs here; inputs are simply passed forward

2. Hidden Layer(s)

- One or more layers between input and output
- Each neuron in a hidden layer:
- Receives inputs from all neurons in the previous layer
- Computes a weighted sum
- Applies a non-linear activation function
- Passes the result to the next layer

3. Output Layer

- Produces the final prediction or classification
- Structure depends on the task:
    - Regression: Often a single neuron with linear activation
    - Binary classification: One neuron with sigmoid activation
    - Multi-class classification: Multiple neurons (one per class) with softmax activation



Structure of an MLP

3. Neurons and Connections

- Each neuron computes a weighted sum of inputs and applies an activation function
- Fully connected between layers (dense connections)

4. Activation Functions

- Introduce non-linearity
- Examples: ReLU, Sigmoid, Tanh, Softmax

5. Loss Function

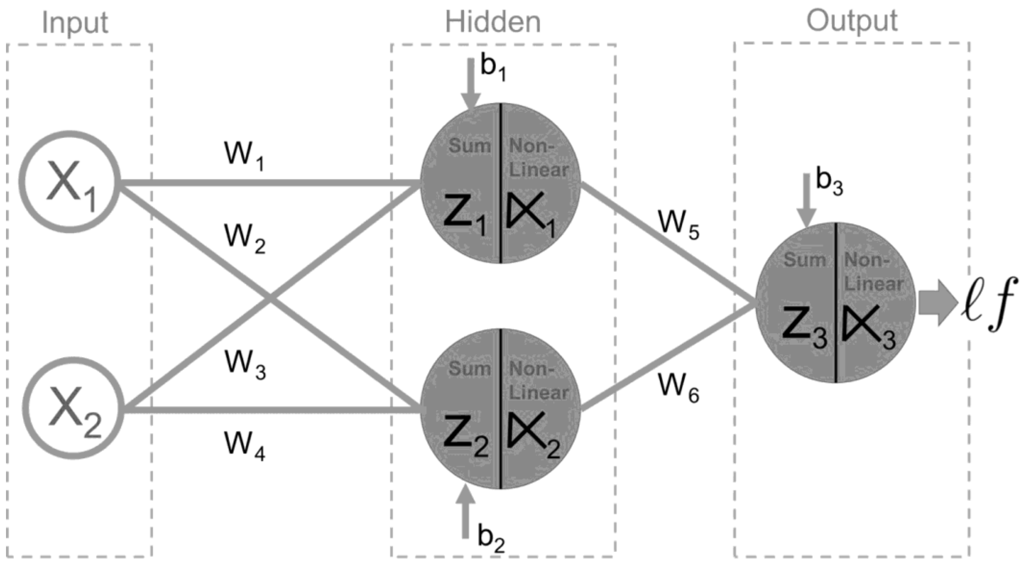
- Measures the error between predicted and true outputs
- Examples: Mean Squared Error, Cross-Entropy

6. Optimizer

- Updates weights to minimize loss
- Examples: SGD, Adam

7. Training Data

- Labeled data used to train the network
- Split into training, validation, and test sets



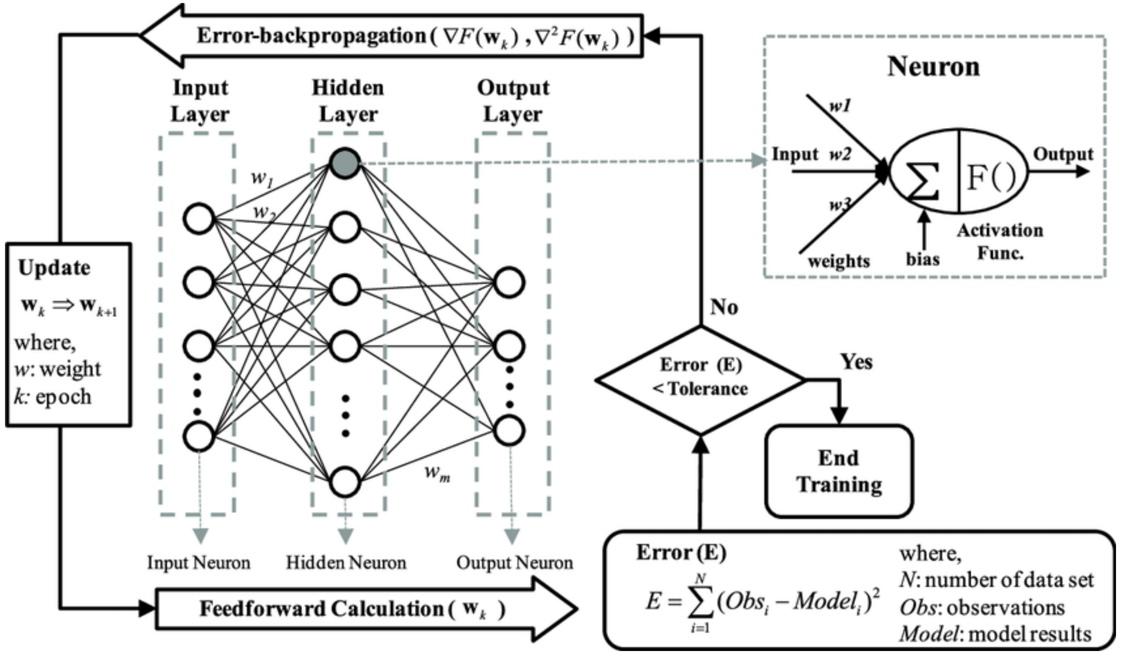
How an MLP Learn – Step by Step

Forward Propagation

- Input features pass through the network layer by layer.
- Each neuron computes a weighted sum of inputs and applies an activation function.
- The final layer produces a prediction.

Loss Computation

- A loss function measures the difference between predicted and actual outputs.
- Common loss functions:
    - Mean Squared Error (regression)
    - Cross-Entropy Loss (classification)



How an MLP Learn – Training Process

Backpropagation

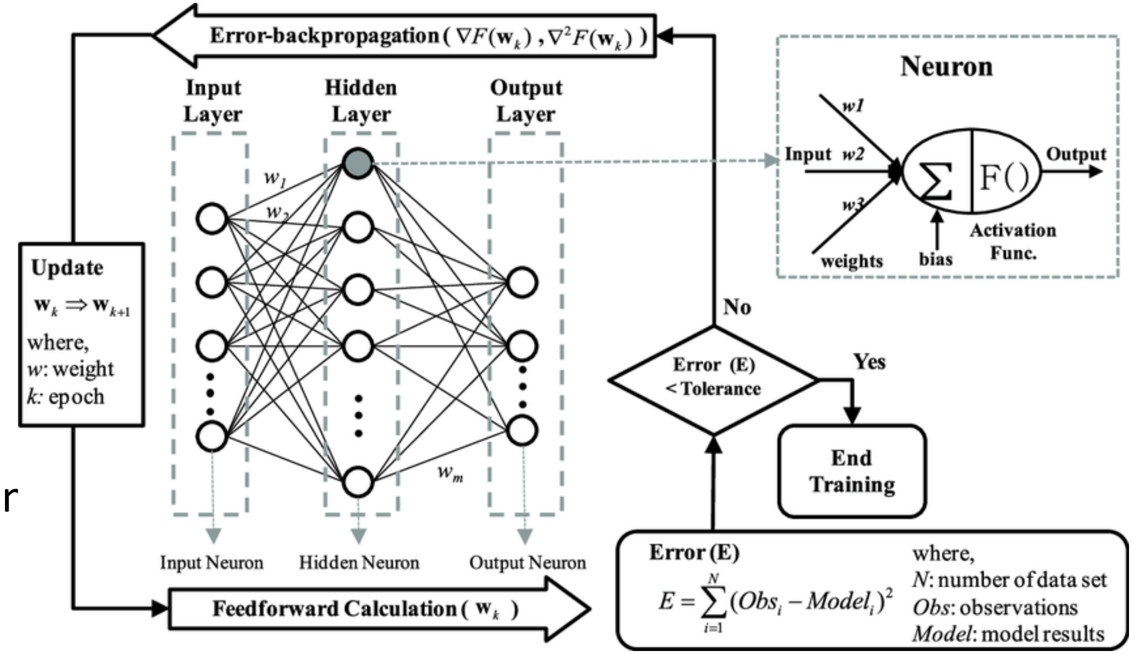
- Gradients of the loss are calculated with respect to each weight using the chain rule.
- This identifies how each weight contributed to the error.

Weight Update

- An optimizer (e.g., SGD) adjusts weights to reduce loss:
  - w := w - learning_rate × gradient
- This process repeats over multiple iterations (epochs) usir training data.

Goal

- Gradually minimize the loss and improve prediction accuracy.
- **Loss Function**: MSE for regression, Cross-Entropy for classification.
- **Optimization**: Backpropagation + Gradient Descent (or Adam).



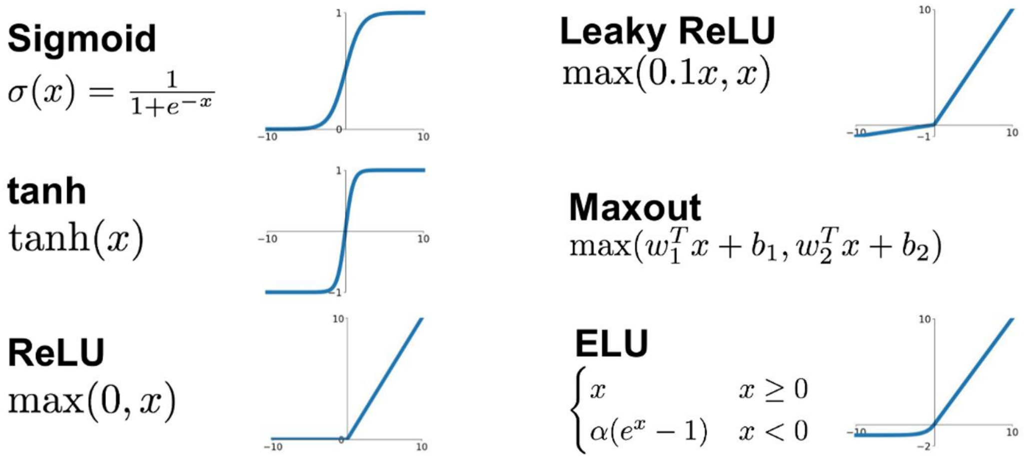
Activation Functions other than Step Function

- Neural Networks use activation functions other than the simple step function in the Perceptron.
- Activation Function helps the neural network use important information while suppressing irrelevant data points (i.e., allows local “gating” of information).

Common Activation Functions: A Toolbox for Neural Networks

- **Sigmoid**: Compresses input to a range between 0 and 1.
- **Tanh**: Like sigmoid but ranges from -1 to 1.
- **ReLU**: Passes positive values, zeroes out negatives.
- **Leaky ReLU**: Allows a small, non-zero gradient for negative inputs.
- **Maxout**: Chooses the most active linear response.
- **ELU**: Smoothly transitions through zero and allows negative outputs.

Each function offers a trade-off between simplicity, flexibility, and computational cost—choose based on the task and depth of your model



https://ml-explained.com/blog/activation-functions-explained

Representing Weights as Matrices

Matrix Representations for Weights

1. **Weight Matrices**:

$W^{(1)}$: Weights from Input to Hidden Layer (shape: input_size × hidden_size):

$[\ w_{1,1}^{(1)}\ w_{1,2}^{(1)}\ ]$
$[\ w_{2,1}^{(1)}\ w_{2,2}^{(1)}]$

Hidden Layer Bias (b_1):
$[\ b_{11}\ b_{12}\ ]$

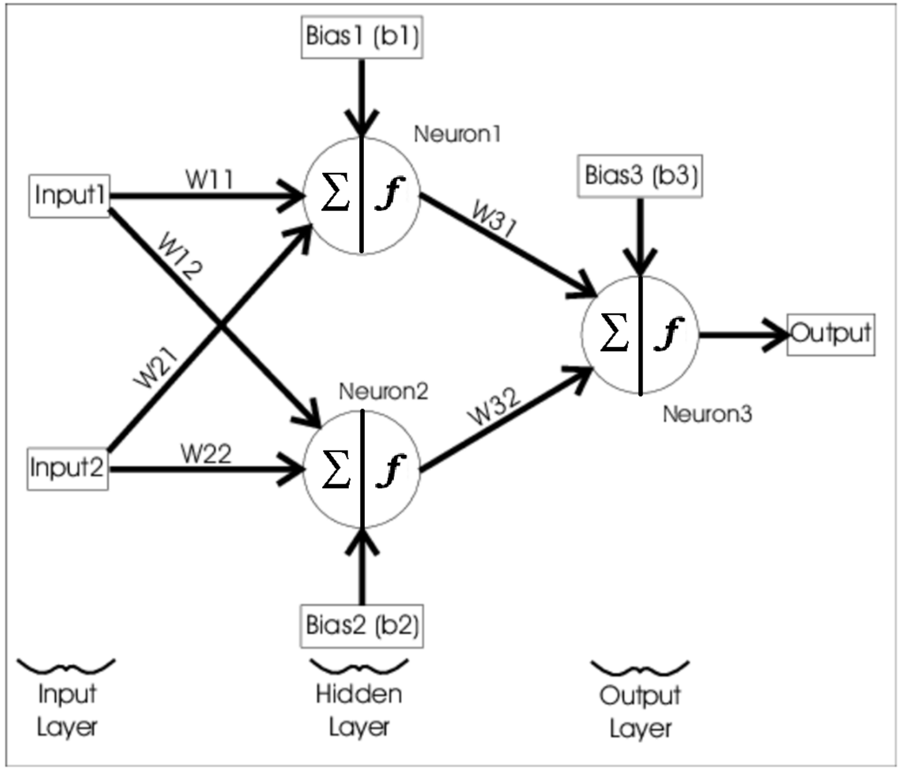
$W^{(2)}$: Weights from Hidden to Output Layer (shape: hidden_size × output_size):

$[\ w_{1,1}^{(2)}\ ]$
$[\ w_{2,1}^{(2)}\ ]$

Output Layer Bias (b_2):
$[\ b_{21}\ ]$

2. **Forward Pass Matrix Operations**:
   - Input to hidden layer: $Z_1 = X \cdot W_1 + b_1$
   - Hidden to output layer: $Z_2 = A_1 \cdot W_2 + b_2$

3. **Backpropagation Matrix Operations**:
   - Weight updates use matrix multiplication between layer activations and error gradients
   - W_2 update: $W_2 \mathrel{+}= A_1^T \cdot \text{delta}_2 \times \text{learning\_rate}$
   - W_1 update: $W_1 \mathrel{+}= X^T \cdot \text{delta}_1 \times \text{learning\_rate}$



The Neural Network Model to solve the XOR Logic (from: https://stopsmokingaids.me/)

Matrix Multiplication

$$\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix} \times \begin{bmatrix}10 & 11\\ 20 & 21\\ 30 & 31\end{bmatrix}$$

$$= \begin{bmatrix}1\text{x}10 + 2\text{x}20 + 3\text{x}30 & 1\text{x}11 + 2\text{x}21 + 3\text{x}31\\ 4\text{x}10 + 5\text{x}20 + 6\text{x}30 & 4\text{x}11 + 5\text{x}21 + 6\text{x}31\end{bmatrix}$$

$$= \begin{bmatrix}10+40+90 & 11+42+93\\ 40+100+180 & 44+105+186\end{bmatrix} = \begin{bmatrix}140 & 146\\ 320 & 335\end{bmatrix}$$

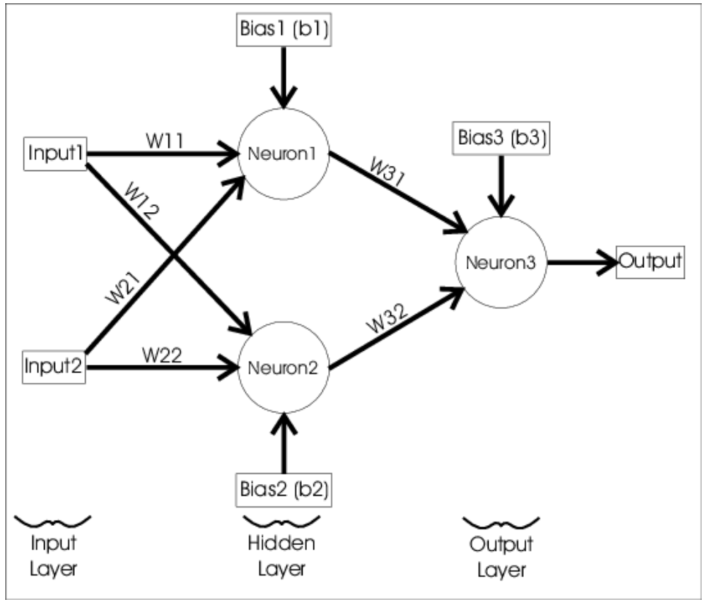
$$\overset{\text{Matrix A}}{\begin{bmatrix}1 & 4 & 6\end{bmatrix}} \bullet \overset{\text{Matrix B}}{\begin{bmatrix}2 & 3\\ 5 & 8\\ 7 & 9\end{bmatrix}}$$

Multi-Layer Perceptron in Python

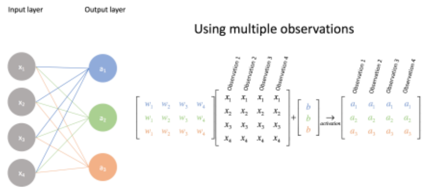
```python
from sklearn.neural_network import MLPClassifier
import numpy as np
# XOR input and output
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 1, 1, 0])
# Define MLP with 1 hidden layer of 2 neurons (minimal config for XOR)
mlp = MLPClassifier(hidden_layer_sizes=(2,), activation='tanh’, solver='adam', learning_rate_init=0.01,
    max_iter=10000, random_state=42)
# Train the model
mlp.fit(X, y)
# Make predictions
predictions = mlp.predict(X)
print("Predictions:\n", predictions)
print("\nWeights (input to hidden):\n", mlp.coefs_[0])
print("\nBias hidden:\n", mlp.intercepts_[0])
print("\nWeights (hidden to output):\n", mlp.coefs_[1])
print("\nBias output:\n", mlp.intercepts_[1])
```

```
Weights (input to hidden):
 [[ 2.7144501   3.27401218]
 [-2.73418453 -3.17014048]]

Bias hidden:
 [ 1.21994174 -1.63451199]
```

```
Weights (hidden to output):
 [[-4.37775211]
 [ 4.46553876]]

Bias output:
 [3.61855675]
```



The Neural Network Model to solve the XOR Logic (from: https://stopsmokingaids.me/)



Open in Colab

Simple Neural Network in Python

```python
import numpy as np

# Simple feedforward neural network
def neural_network(x, weights):
    # First hidden layer with ReLU activation
    hidden = np.maximum(0, np.dot(x, weights[0]) + weights[1])  # ReLU activation
    # Output layer
    output = np.dot(hidden, weights[2]) + weights[3]
    return output

# Example network with random weights
input_size = 3
hidden_size = 4
output_size = 2

# Initialize random weights
W1 = np.random.randn(input_size, hidden_size)  # Input → Hidden
b1 = np.random.randn(hidden_size)              # Hidden bias
W2 = np.random.randn(hidden_size, output_size) # Hidden → Output
b2 = np.random.randn(output_size)              # Output bias
weights = [W1, b1, W2, b2]

# Example input
x = np.array([0.5, 0.3, 0.2])

# Forward pass
prediction = neural_network(x, weights)
print(f"Network prediction: {prediction}")
```

Open in Colab

Backpropagation (Conceptual)

```python
import numpy as np

# Simplified backpropagation example
def train_step(x, y_true, weights, learning_rate=0.01):
    # Forward pass
    hidden = np.maximum(0, np.dot(x, weights[0]) + weights[1])  # ReLU
    y_pred = np.dot(hidden, weights[2]) + weights[3]

    # Compute loss (Mean Squared Error)
    loss = np.mean((y_pred - y_true)**2)

    # Backpropagation (simplified)
    # Output layer gradients
    grad_y_pred = 2 * (y_pred - y_true) / len(y_true)
    grad_W2 = np.dot(hidden.T, grad_y_pred)
    grad_b2 = np.sum(grad_y_pred, axis=0)

    # Hidden layer gradients
    grad_hidden = np.dot(grad_y_pred, weights[2].T)
    grad_hidden[hidden <= 0] = 0  # ReLU gradient
    grad_W1 = np.dot(x.T, grad_hidden)
    grad_b1 = np.sum(grad_hidden, axis=0)

       # Update weights
    weights[0] -= learning_rate * grad_W1
    weights[1] -= learning_rate * grad_b1
    weights[2] -= learning_rate * grad_W2
    weights[3] -= learning_rate * grad_b2

       return loss, weights

# Example usage
# (In practice, we would use frameworks like PyTorch or TensorFlow)
```

Open in Colab

The Difference Between a Perceptron and an MLP?

Feature	Perceptron	Multilayer Perceptron (MLP)
Layers	Only 1 layer (no hidden layers)	2+ layers (has hidden layers)
Activation Function	Step function (hard threshold)	Nonlinear (e.g., ReLU, sigmoid, tanh)
Learning Rule	Simple rule: update on error	Gradient descent + backpropagation
Tasks It Can Solve	Only linearly separable problems	Nonlinear, complex problems