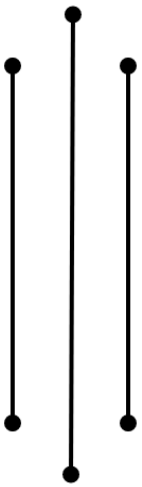
Tribhuvan University
Institute of Engineering
Thapathali Campus, Thapathali

Artificial Intelligence



Submitted by:
Jenish Pant (THA078BEI018)
Sajen Maharjan (THA078BEI048)

Submitted to:
Department of Electronics and Computer Engineering

May 26, 2025

Title

Implementation of Perceptron Learning Algorithm for Logic Gate Simulation

Abstract

This experiment demonstrates the implementation of the Perceptron Learning Algorithm (PLA) to simulate the behavior of a basic AND logic gate. The perceptron, a foundational model in artificial neural networks, was trained using supervised learning on labeled input-output pairs. The learning process involved iterative weight adjustments via error-correction until convergence. Results confirm that the perceptron accurately models the AND gate, validating its capability for linearly separable binary classification tasks. This report details the methodology, implementation, results, and theoretical implications of PLA in machine learning.

Introduction

Artificial Neural Networks (ANNs) are computational models inspired by biological neural networks. The perceptron, introduced by Rosenblatt in 1958, represents the simplest form of an ANN and serves as a binary classifier. The Perceptron Learning Algorithm (PLA) adjusts synaptic weights based on prediction errors, enabling the model to learn from training data. Logic gates such as AND, OR, and NOT provide an ideal framework for studying perceptrons due to their simplicity and well-defined behavior. This experiment focuses on implementing PLA to simulate the AND gate, a linearly separable function, thereby illustrating the fundamental principles of supervised learning in neural networks.

Experimentation/Model

The implemented model is a single-layer perceptron with two input neurons (x_1, x_2) and one output neuron (y). The perceptron uses a sigmoid activation function to introduce non-linearity, defined as:

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

where $z = w_1 x_1 + w_2 x_2 + b$, with w_1, w_2 as the input weights and b the bias. Two versions of the perceptron were tested:

1. **Hardcoded Perceptron:** This version uses fixed weights $w_1 = w_2 = 7.27$ and bias $b = -11.055$. The model predicts the AND gate output by computing the weighted sum and passing it through the sigmoid function. The final output is determined by thresholding the sigmoid output at 0.5.

2. **Trainable Perceptron:** This model learns the weights and bias using the Perceptron Learning Algorithm. The training dataset contains four input-output pairs for the logical AND gate. The weights are initialized randomly and updated iteratively using gradient descent:

   $$w_i \leftarrow w_i + \eta \cdot (y - \hat{y}) \cdot x_i, \quad b \leftarrow b + \eta \cdot (y - \hat{y}),$$

   where η is the learning rate, y is the true output, and $\hat{y}$ is the predicted output obtained from the sigmoid activation. Training proceeds over multiple epochs until the model converges.

Both models demonstrate the ability of the perceptron to model linearly separable logic functions such as the AND gate.

Training Algorithm

The implemented training procedure is based on the Perceptron Learning Algorithm (PLA), adapted to use a sigmoid activation function. The steps are as follows:

1. **Initialization:** Weights w_1, w_2 and bias b are initialized randomly within the range $[-1, 1]$.
2. **Training Loop:** For a fixed number of epochs:
   - For each input-output pair $(\mathbf{x}, d)$ in the training set:
     (a) Compute the weighted sum:
     $$z = w_1 x_1 + w_2 x_2 + b$$
     (b) Apply the sigmoid activation function:
     $$\hat{y} = \frac{1}{1 + e^{-z}}$$
     (c) Compute the error:
     $$e = d - \hat{y}$$
     (d) Update the weights and bias:
     $$w_i \leftarrow w_i + \alpha \cdot e \cdot x_i \quad \text{(for } i = 1, 2)$$
     $$b \leftarrow b + \alpha \cdot e$$
     where α is the learning rate.
3. **Termination:** The algorithm runs for a predefined number of epochs (e.g., 1000). Since the sigmoid function outputs continuous values, exact convergence may not occur, but the output will approximate the desired logic behavior.

Result

The perceptron was trained on the AND gate dataset using a sigmoid activation function and a simple weight update rule. Key observations from the experiment are as follows:

- The model was trained for 1000 epochs with a learning rate $\alpha = 0.01$.
- The weights and bias converged gradually and approximated values close to those that correctly model the AND gate.
- After training, the final predictions were thresholded at 0.5 to produce binary outputs.
- The trained perceptron produced the following outputs:

x_1	x_2	Output
0	0	0
0	1	0
1	0	0
1	1	1

Discussion

The experiment validates that a single-layer perceptron can model linearly separable functions such as the AND gate. Key findings include:

- The learning rate α significantly impacts convergence speed. Smaller values ($\alpha < 0.1$) resulted in slower learning, while excessively large values risked unstable updates.
- Random weight initialization affected training dynamics, but did not hinder convergence due to the simplicity and linear separability of the AND gate.
- The perceptron's inability to solve non-linearly separable functions (e.g., XOR) highlights a fundamental limitation of single-layer models, motivating the study of multi-layer perceptrons and more advanced architectures.

Conclusion

This experiment confirms the effectiveness of the Perceptron Learning Algorithm for binary classification tasks involving linearly separable data. The successful simulation of the AND gate emphasizes the perceptron's role as a foundational unit in neural network models. While the simplicity of the model limits its applicability to complex tasks, it serves as a valuable introduction to concepts like weight updates, activation functions, and convergence. Future work could explore multi-layer perceptrons and backpropagation algorithms to extend the model's capability to non-linear problems like XOR. The insights gained here provide an essential stepping stone toward understanding deep learning systems.

Appendix A: Source Code

1. Hardcoded Perceptron with Sigmoid Activation

```c
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

#define euler 2.718281828

float sigmoid(float x)
{
    return (1 / (1 + pow(euler, -x)));
}

bool andPerceptron(bool x1, bool x2)
{
    float weightedSum = 7.27 * x1 + 7.27 * x2 - 11.055;
    return (sigmoid(weightedSum) > 0.5 ? 1 : 0);
}

bool unitStep(int x)
{
    return (x > 0 ? 1 : 0);
}

int main()
{
    printf("AND TABLE\n");
    printf("%d\n", andPerceptron(0, 0));
    printf("%d\n", andPerceptron(0, 1));
    printf("%d\n", andPerceptron(1, 0));
    printf("%d\n", andPerceptron(1, 1));

    printf("Unit Step\n");
    printf("%d\n", unitStep(1));

    printf("Sigmoid\n");
    printf("%f\n", sigmoid(0));
}
```

2. Trainable Perceptron Using PLA with Sigmoid

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define training_samples 4

typedef float sample[3];
sample and_dataset[training_samples] =
{
    {0, 0, 0},
    {0, 1, 1},
    {1, 0, 0},
    {1, 1, 1},
};
sample *train = and_dataset;

float randf() { return (2*((float)rand() / (float)RAND_MAX) - 1 ); }
float sigmoidf(float x) { return 1.f / (1.f + expf(-x)); }

void init_weights(float *w0, float *w1, float *bias)
{
    *w0 = randf();
    *w1 = randf();
    *bias = randf();
}

float perceptron(float w0, float w1, float bias, unsigned row)
{
    float perceptron = w0 * train[row][0] + w1 * train[row][1] +
       bias * 1;
    perceptron = sigmoidf(perceptron);
    return perceptron;
}

int main()
{
    srand(time(0));
    float w0, w1, bias;
    init_weights(&w0, &w1, &bias);

    float learning_rate = 1e-2;
    size_t epochs = 1e3;

    for (size_t j = 0; j < epochs; j++)
    {
        for (size_t i = 0; i < training_samples; i++)
        {
            float y = train[i][2];
```

```
            float error = y - perceptron(w0, w1, bias, i);

            w0 += learning_rate * error * train[i][0];
            w1 += learning_rate * error * train[i][1];
            bias += learning_rate * error;

            printf("w0 = %6.6f, w1 = %6.6f, bias = %6.6f\n", w0,
               w1, bias);
        }
    }

    printf("\n\t\tAND gate\nw0 = %6.6f, w1 = %6.6f, bias =
       %6.6f\n\n", w0, w1, bias);
    printf("%10s %10s %10s\n", "Input[0]", "Input[1]", "Output");
    for (size_t i = 0; i < training_samples; ++i)
    {
        printf("%10.6f %10.6f %9d\n", train[i][0], train[i][1],
           perceptron(w0, w1, bias, i) > 0.5 ? 1 : 0);
    }
}
```