University of Missouri–St. Louis

Department of Computer Science

A Project Report on

IMPLEMENTATION OF OR GATE USING SIMULATED ANNEALING

Submitted by:
Sachina Koirala
Student ID: 18281333

Submitted to:
Dr. Badri Adhikari

October 4, 2024

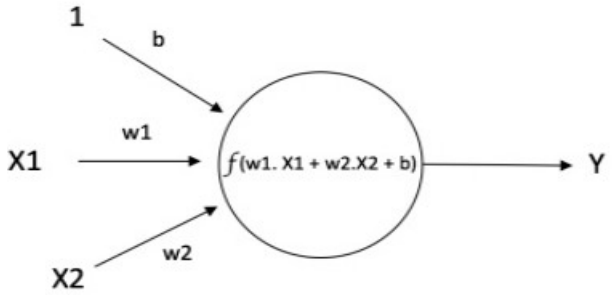
1 Introduction

Simulated annealing (SA) is a probabilistic technique used to approximate the global optimum of a given function. The search process begins with a high-energy state, representing an initial solution, and gradually lowers the temperature, a control parameter, until it reaches a state of minimum energy, corresponding to the optimal solution. In the context of neural networks, SA plays a crucial role in optimizing weight configurations, enabling models to learn more effectively from data.

This report will discuss the implementation of simulated annealing to address the OR gate optimization problem, which is essential for understanding how artificial neurons can perform logical operations.

2 Background

The OR gate is a basic digital logic gate that outputs true or 1 if at least one of the inputs is true. The objective of this project is to optimize the weights of the neuron that implements this function using the Simulated Annealing approach.



Output of neuron = Y= f(w1. X1 + w2.X2 + b)

3 Problem Statement

The aim of this project is to train an artificial neuron to act like an OR gate by adjusting its weights. The neuron takes two binary inputs (either 0 or 1), and its output is a value between 0 and 1, simulating the behavior of an OR logic gate. To do this, we will use the simulated annealing algorithm, an optimization method that helps find the best set of weights (Wx, Wy, and Wb) by trying different values and gradually improving them.

In this project, the sigmoid function is used to process the input values. It takes the weighted sum of the inputs and converts it into a value between 0 and 1, allowing for a smoother, more flexible output that can be compared to the expected result. The goal is to adjust the weights in such a way that minimizes the error between the neuron's output and the actual OR gate output, effectively training the neuron to mimic the OR function.

The simulated annealing algorithm starts by exploring random weights and then refines them step-by-step until we find the set of weights that best fit the OR gate's logic.

4 Goals and Objectives

- To track the optimization process of the weights using Google Sheets.
- To develop and implement the simulated annealing algorithm in a python programming language to find the best weights.
- To evaluate and compare how different temperature schedules impact the optimization process.

5 Code Implementation

Google sheet link: Tracing the SA algorithm for finding the weights in a neural representing a boolean OR gate

Python Code on Google Colab : Code Implementation

For the code implementation following tools were used:

- Visual Studio Code
- Google Collaboratory
- Programming Language: Python

The following Python code implements the simulated annealing algorithm to optimize the weights for an OR gate using a sigmoid activation function:

```
import numpy as np
import random
import matplotlib.pyplot as plt

# Constants for the problem
INITIAL_WEIGHTS = np.array([0, 0, 0])  # values for Wx, Wy, Wb
OR = [0, 1, 1, 1]
XY = [[0, 0, 1], [0, 1, 1], [1, 0, 1], [1, 1, 1]]

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

# Define the error function based on the sigmoid formula
def calculate_error(weights):
    total_error = 0
    for i in range(len(OR)):
        value = np.dot(XY[i], weights)  # X * Wx + Y * Wy + 1 *
           Wb
        prediction = sigmoid(value)
        error = np.abs(OR[i] - prediction)
        total_error += error
    return total_error
```

```
# Temperature scheduling function with division by 1.2
def schedule(T):
    return T / 1.2  # Temperature: T.Next =  T.current/1.2

# Function to create a random neighbor (Current config +
   RANDBETWEEN(-2, 2))
def make_node(current):
    # Generate random changes for each weight component Wx, Wy,
       and Wb
    random_change = np.array([random.randint(-2, 2), random.
       randint(-2, 2), random.randint(-2, 2)])
    return current + random_change

# Simulated Annealing function with temperature decrease and
   iteration tracking
def simulated_annealing(problem, initial_temperature):
    current = INITIAL_WEIGHTS.copy()
    current_value = calculate_error(current)

    # Initial temperature
    T = initial_temperature

    # Lists to track temperatures and errors
    temperatures = []
    iterations = []
    error_history = []
    delta_E_values = []

    tolerance = 1e-3  # Convergence criterion for error change

    for t in range(1, 1000):
        if T < tolerance:  # Stop if temperature is too low
            return current, temperatures, iterations,
               error_history, delta_E_values

        next_node = make_node(current)
        next_value = calculate_error(next_node)
        delta_E = next_value - current_value

        if delta_E < 0:  # Accept the move if error decreases
            current = next_node
            current_value = next_value
        else:  # Accept with probability based on temperature
            acceptance_probability = np.exp(-delta_E / T)
            if random.random() < acceptance_probability:
                current = next_node
                current_value = next_value

        # Update temperature based on the new schedule
        T = schedule(T)
```

```
            # Track temperature, iteration, and error history
            temperatures.append(T)
            iterations.append(t)
            error_history.append(current_value)
            delta_E_values.append(delta_E)

    return current, temperatures, iterations, error_history,
        delta_E_values

# Function to plot results
def plot_results(error_history, temperatures, iterations,
    delta_E_values, title):
    plt.figure(figsize=(15, 10))

    # Plot Temperature vs Iterations
    plt.subplot(3, 1, 3)
    plt.plot(iterations, temperatures, color='blue')
    plt.title(f'Temperature vs Iterations ({title})')
    plt.xlabel('Iterations')
    plt.ylabel('Temperature')

    # Plot Delta E vs Iterations
    plt.subplot(3, 1, 1)
    plt.plot(range(len(delta_E_values)), delta_E_values, color='
        red')
    plt.title(f'Delta E vs Iterations ({title})')
    plt.xlabel('Iterations')
    plt.ylabel('Delta E')
    plt.axhline(0, color='black', lw=0.5, ls='--')  # Line at
        zero for reference

    # Plot Objective Function vs Iterations
    plt.subplot(3, 1, 2)
    plt.plot(range(len(error_history)), error_history, color='
        green')
    plt.title(f'Objective Function vs Iterations ({title})')
    plt.xlabel('Iterations')
    plt.ylabel('Objective Function Value')
    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    initial_temp = 10  # Starting temperature
    # Run the simulated annealing algorithm with an initial
        temperature of 10
    final_solution, temperatures, iterations, error_history,
        delta_E_values = simulated_annealing(None, 10)
    final_error = calculate_error(final_solution)

    # Print results
```

```
    print(f"The final solution weights: {final_solution}")
    print(f"Number of iterations: {iterations[-1]}")
    print(f"Final temperature: {temperatures[-1]}")
    print(f"Final delta E value: {delta_E_values[-1]}")

    # Plot the results with a title
    plot_results(error_history, temperatures, iterations,
       delta_E_values, "Simulated Annealing Results")
```

6 Output

By running the above code in Python using three different subsequent temperature schedules, we can observe the following results and also the graph is plotted accordingly:

1. **Temperature Schedule: Tnext=Tcurrent/1.2**

```
PS C:\Users\sachi\Desktop\First sem\AI> python project.py
------------------------------
The final solution weights: [11 11 -5]
Number of iterations: 51
Final temperature: 0.0009157068259764378
Final delta E value: 0.007018773647248231
```

Figure 1: Output for 1.2 Schedule

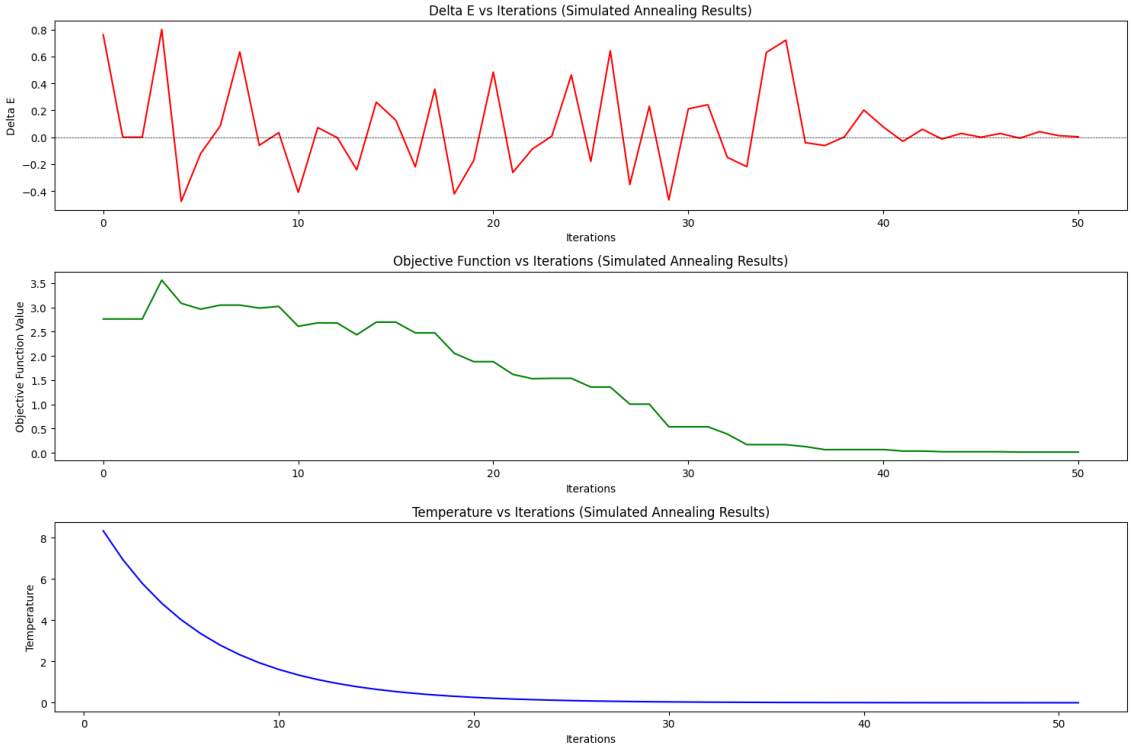


Figure 2: Graph at Temperature schedule T/1.2

This schedule cools the temperature moderately.There is a balance between exploring new solutions and refining existing ones, with a low error (final delta E near

zero).The number of iterations is moderate which is 51, meaning it converges in a reasonable time while still finding a strong solution.

2. **Temperature Schedule: Tnext=Tcurrent/1.9**

```
PS C:\Users\sachi\Desktop\First sem\AI> python project.py
------------------------------
The final solution weights: [2 7 2]
Number of iterations: 15
Final temperature: 0.000658712622608394
Final delta E value: 0.08342805795424024
```

Figure 3: Output for 1.9 Schedule

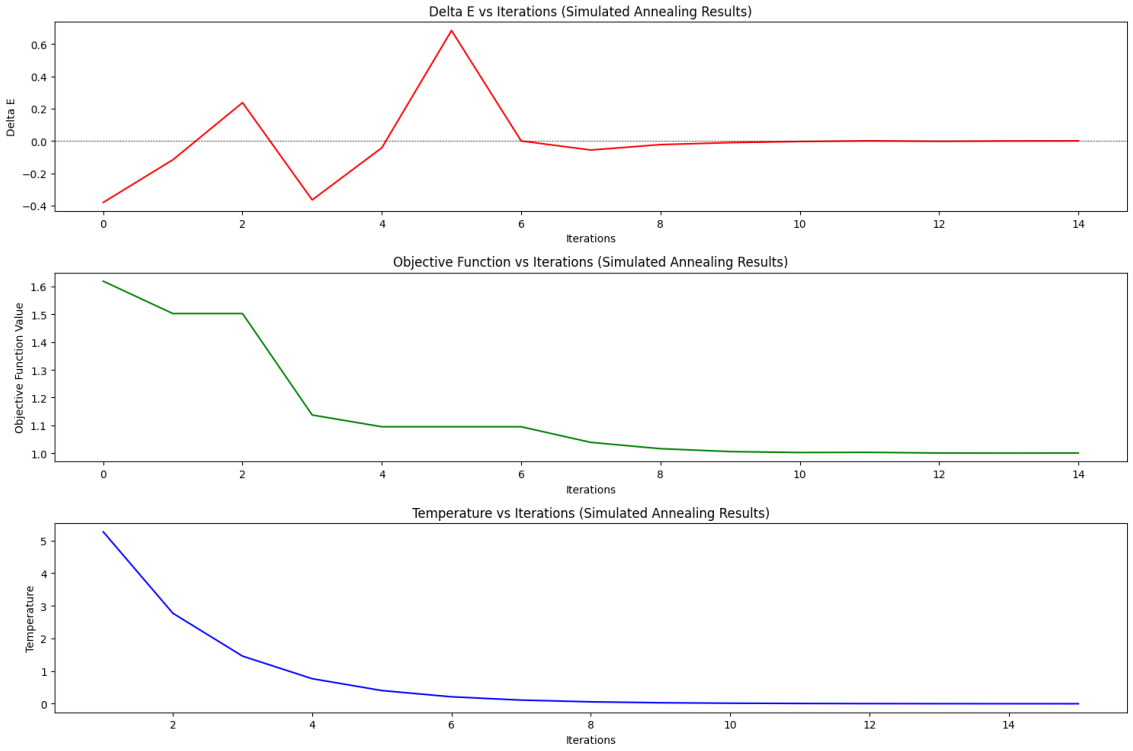


Figure 4: Graph at Temperature schedule T/1.9

Compared to 1.2 we can see that the number of iteration to reach the optimal solution has decreased, which means the faster cooling rate here results in fewer iterations. The temperature drops quickly, which limits the algorithm's ability to explore many potential solutions.As a result, the solution found is less optimal, with a higher final delta E, indicating a larger error in the final solution.

3. **Temperature Schedule: Tnext=Tcurrent/1.1**

```
PS C:\Users\sachi\Desktop\First sem\AI> python project.py
------------------------------
The final solution weights: [13 12 -6]
Number of iterations: 97
Final temperature: 0.0009658496786487173
Final delta E value: -0.010002134429899834
```

Figure 5: Output for 1.1 Schedule

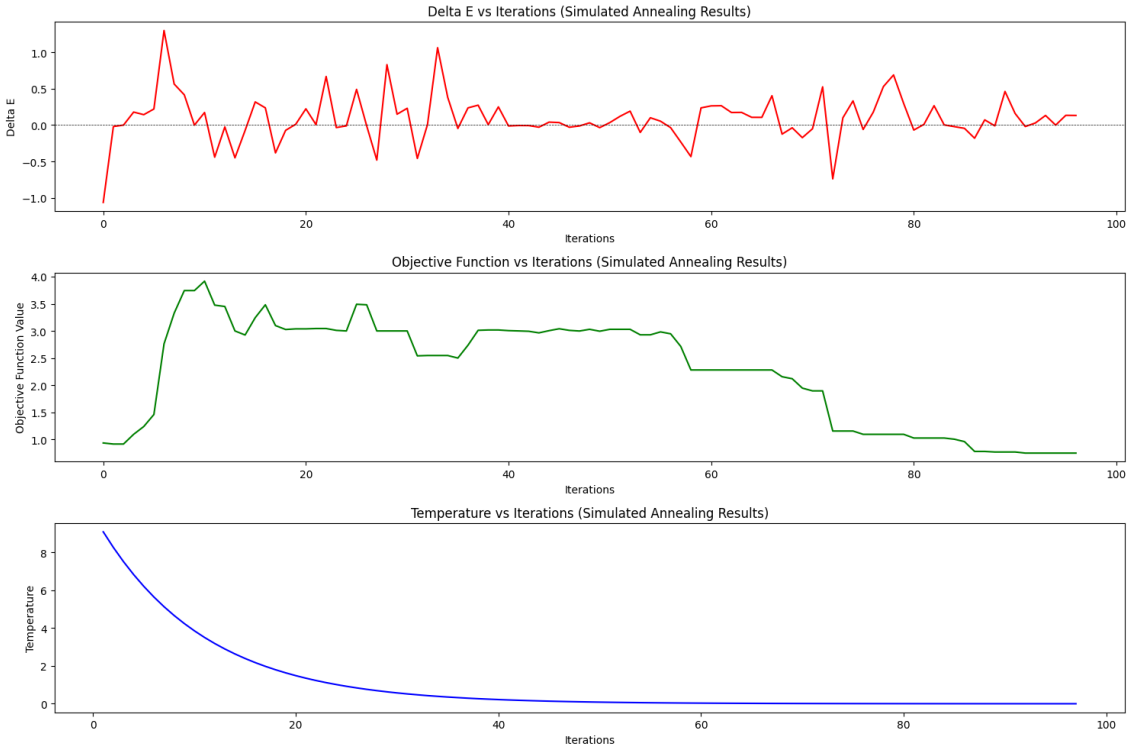


Figure 6: Graph at Temperature schedule T/1.1

From the above graph and output, we can observe that the number of iteration for 1.1 is comparatively more as than that of 1.2 and 1.9.This means it takes longer to converge, but the final solution is the most optimal with the lowest error and the negative delta E indicates effective minimization.It is the slowest cooling rate that allows the algorithm to search the space more.

7 Challenges faced

While doing this project, the challenges faced are as follows:

7.1 Randomness in value

Because simulated annealing is stochastic in nature,it was challenging to get the consistent results with each execution due to its random elements. This variation made it challenging to evaluate the impact of various temperature schedules effectively.

7.2 Choosing Temperature schedules

Initially, I tried several values for the cooling rate, aiming to find schedules that would represent slow (T/1.1), moderate (T/1.2), and fast (T/1.9) cooling. It was challenging to balance the rates in a way that allowed me to compare their effects clearly on the optimization process.

7.3 Temperature value

The algorithm described in the book suggests that iterations should continue until the temperature reaches zero (T = 0). However, this approach can lead to excessive runtime, as the algorithm may run for an extended period while the temperature approaches zero, resulting in an impractically high number of iterations. To address this challenge, a tolerance threshold is introduced (set at 1e-3) so that if the temperature drops below it, the loop terminates early.

8 Conclusion

In conclusion, this project demonstrated how simulated annealing can be used to optimize an artificial neuron that replicates the behavior of a boolean OR gate. By testing various temperature schedules, it was found that different cooling rates affect the optimization process. A slower cooling rate (T/1.1) led to more accurate results, although it required more iterations to achieve those results. In contrast, a faster cooling rate (T/1.9) allowed for quicker convergence but did not yield the best possible outcomes.

9 References

Introduction-to-a-powerful-optimization-technique-simulated-annealing

Tracing the simulated annealing algorithm to optimize the weights of an AND gate artificial neuron- Professor Badri Adhikari

Google sheet UMSL : Hill climbing and simulated annealing for finding the weights for a boolean OR gate

Grammarly: An online tool that helps with grammar, punctuation, and style

Artificial-Intelligence-A-Modern-Approach-4th-Edition https://www.grammarly.com/