Charles de Souza
Automation & Control Engineering's student

Simulation of 1 Artificial Neuron in a Neural Network using Python

Campinas, São Paulo
2024

Condensed Theory

Difficult to start talking about this subject without making an analogy between our biological neurons and the artificial neurons given that it is a cornerstone concept in the field of artificial intelligence, something I'm passionate about, and the neural networks. Just as biological neurons process and transmit signals in the human brain, artificial neurons or perceptrons emulate this behavior to perform computational tasks. Simply put, both types of neurons serve as basic processing units, receiving input signals, integrating them using ~~mathemagics~~ mathematics, and producing output signals. Just like the biological neuron "fires" when the energy inside it increases, the artificial one also has a similar mechanism.

The firing of an artificial neuron, a fundamental concept in neural networks, parallels the behavior of biological neurons. In artificial neural networks, the neuron fires when the weighted sum of its inputs, along with a bias term if present, surpasses a certain threshold. This threshold is often determined by an activation function, which transforms the neuron's input into an output signal. Common activation functions include the step function, sigmoid function, and rectified linear unit (ReLU).

When the neuron fires, it produces an output signal that propagates through the network, influencing the activation of subsequent neurons. This propagation of signals enables the network to perform complex computations and learn from data. The firing of neurons, governed by activation functions and adjustable parameters such as weights and biases, underpins the neural network's ability to process information and make predictions.

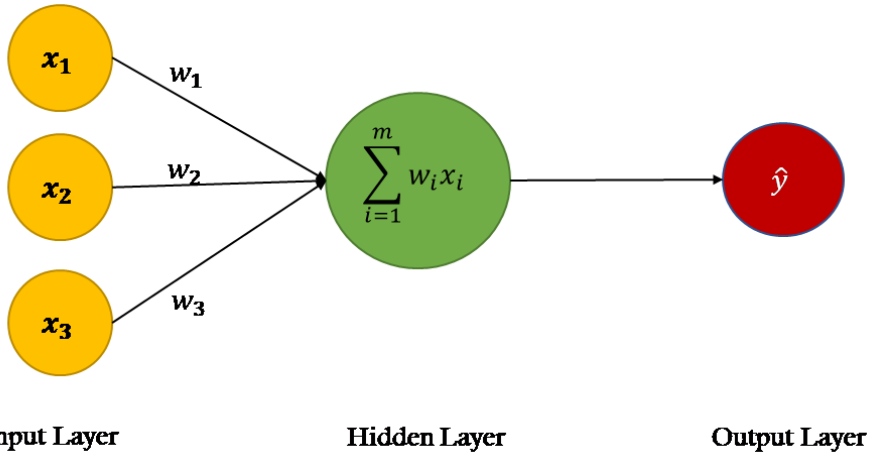

Figure 1 – Basic Unit of an Artificial Neural Network (Artificial Neuron)

In the realm of deep learning, even with just a single neuron, the underlying theory revolves around supervised learning. Here in the figure above, the neuron

receives input signals weighted by adjustable parameters, known as input weights. These weighted inputs are then processed through an activation function inside the “Hidden Layer”, producing an output signal. Through an iterative process called training, the neuron adjusts its input weights using a learning algorithm, often gradient descent, to minimize the difference between its output and the desired output. By continuously refining its input weights based on the error between actual and desired outputs, the neuron gradually learns to approximate intricate input-output relationships, showcasing the transformative potential of neural networks in learning from data and making accurate predictions upon making the error equals to 0.

Goals

In this personal project, the goal is to apply the theory regarding the Neural Network (Deep Learning), using only one neuron to simplify. The aim is to understand how the neural network adapts and corrects errors during the learning process. This personal project came as a curiosity while I was studying the theory about the fundamental concepts of neural networks, with the intention of gaining insights into their behavior and their potential applications in more complex scenarios by aligning the theory with the real life.

Materials and Methods

In this personal exploration of deep learning fundamentals and using Python, I simulated a single neuron to correct its error given the input, the output desired, the input weight and the learning rate. These four parameters and their respectives values were used in the training process, in which each iteration brought the neuron closer and closer to the output desired with different inputs.

Through a series of calculations and activation functions, I observed the neuron's output and analyzed the error between its prediction and the desired outcome. By adjusting input weights based on this error and the learning rate, the neural network gradually honed its abilities and eventually learning in the end of each process. This project provided invaluable insights into the mechanisms of neural networks and the “magic” of deep learning.

Results

In exploring the behavior of a single neuron within a neural network, various scenarios are examined to understand its responsiveness to different input signals. Here's the code to follow along:

Neural_Network.py

ai-dev > Neural_Network.py > ...

```python
# Simulation of 1 neuron in a Neural Network to understand "deeper" the basics of Deep Learning.

import math # Declaring initial variables for the simulation.

input = 0 # Setting the initial input.
output_desired = 1 # Desired output for learning.

input_weight = 0.5 # Initial weight assigned to the input.
learning_rate = 0.1 # Learning rate for adjusting weights.

# Activation function to determine neuron output.
def activation(sum):
    if sum >= 0:
        return 1
    else:
        return 0

print("Input: ", input, "\nOutput desired: ", output_desired)

error = math.inf  # Any number multiplied by zero is zero, this short block of code doesn't return 1 (line 14) for this case, but it returns 0 instead.
virtual_neuron = 1 # Virtual neuron that always has as result 1 for learning purposes.
virtual_neuron_weight = 0.5 # Initial weight assigned to the virtual neuron.

counter_iterations = 0 # Counter to track the number of iterations.

# Loop to train the neuron until the error is 0.
while not error == 0:
    counter_iterations += 1 # Incrementing the iteration counter.
    print("\nIteration ", counter_iterations, ":", sep='') # Showing the number of iterations.

    # Calculating the sum of inputs multiplied by their respective weights.
    sum = (input * input_weight) + (virtual_neuron * virtual_neuron_weight)

    output = activation(sum) # Applying activation function to get the neuron output.

    print("Output: ", output) # Displaying the neuron's output.

    error = output_desired - output # Calculating the error.

    print("Error: ", error) # Displaying the current error.

    # Updating weights using the error and learning rate.
    if not error == 0:
        input_weight = input_weight + (learning_rate * input * error)
        virtual_neuron_weight = virtual_neuron_weight + (learning_rate * virtual_neuron * error) # Adjusting weights to prevent divergence.

print("\nThe neural network has learned successfully!!!") # Message of success!
```

Figure 2 – Code's printscreen

For instance, when both the input and desired output are zero, it reveals how the neuron behaves in the absence of an input signal, potentially resulting in an output of zero if properly adjusted:

```
5   input = 0 # Setting the initial input.
6   output_desired = 0 # Desired output for learning.
7
8   input_weight = 0.5 # Initial weight assigned to the input.
9   learning_rate = 0.1 # Learning rate for adjusting weights.
```

Figure 3 – Initial parameters and values

For these specific inputs, the neuron is expected to produce an output of zero when the input signal is zero. However, the original code initializes the error variable

to positive infinity, which leads to incorrect behavior during the training process.

To address this issue, the code was modified to properly handle the case where the desired output is zero. By setting error to positive infinity, the condition while not error == 0 was never met, preventing the training loop from executing. Consequently, the neuron's weights were not updated, and the network failed to learn the right behavior.

By initializing error to zero and adjusting the logic of the training loop, the code now accurately represents the scenario where the desired output is zero. This ensures that the neuron can effectively learn to produce the desired output when the input is zero, here below are the iterations necessary to make the error equals to 0:

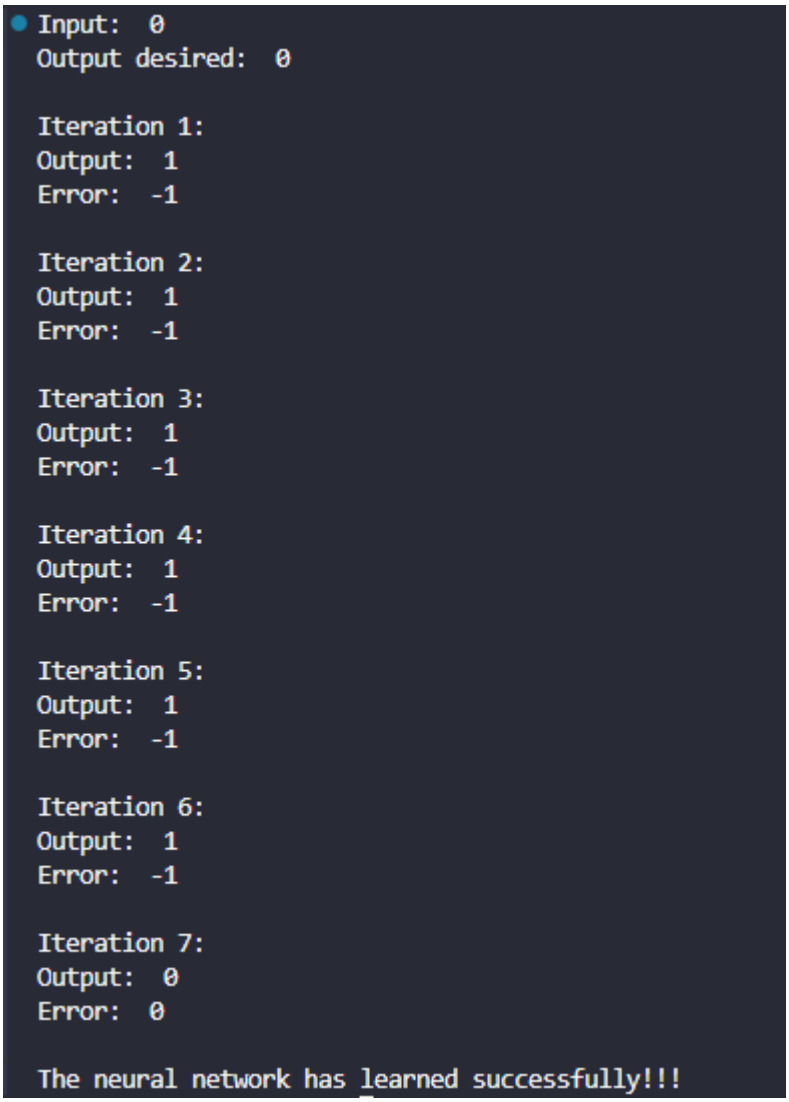

Figure 4 – First example

Now, for both input and output equal to 1:

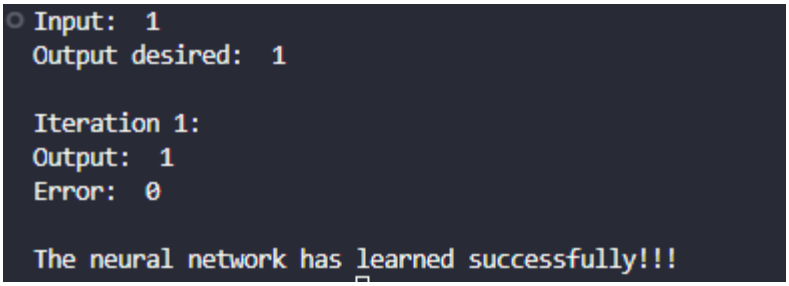

Figure 5 – Second example

Here, the neuron's output is computed based on the input, its weights, and the activation function. If the input weight is appropriately adjusted and the activation function thresholds the sum to one for positive inputs, the neuron should produce the desired output of one and, of course, the error of zero.

For input = 0 and output = 1:

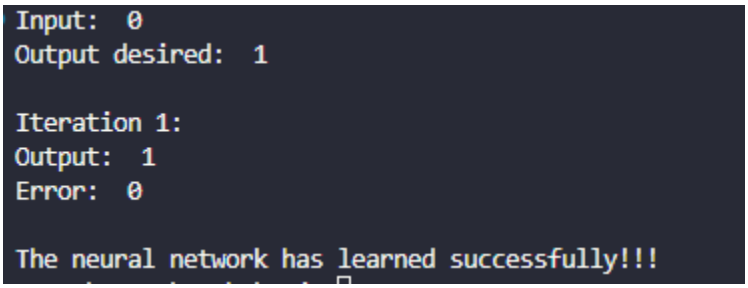

Figure 6 – Third example

This is interesting as well, because it suggests a situation where the neuron is expected to produce an output signal even without receiving any input signal. The code attempts to adjust the neuron's weights during the learning process to match the desired output. However, if the neuron is unable to learn the correct behavior, it may produce an output of one even when the input is zero, leading to a mismatch between the actual and the desired outputs, but learning accordingly with one only iteration.

For input = 1 and output = 0:

```
Input:  1
Output desired:  0

Iteration 1:
Output:  1
Error:  -1

Iteration 2:
Output:  1
Error:  -1

Iteration 3:
Output:  1
Error:  -1

Iteration 4:
Output:  1
Error:  -1

Iteration 5:
Output:  1
Error:  -1

Iteration 6:
Output:  1
Error:  -1

Iteration 7:
Output:  0
Error:  0

The neural network has learned successfully!!!
```

Figure 7 – Fourth example

Here, it took the learning process 7 iterations by adjusting the neuron's weights during the learning process to achieve the desired behavior. However, if the neuron fails to properly adjust its weights or if the activation function does not threshold the sum appropriately, it can produce an incorrect output of zero when the input is one, resulting in a mismatch with the desired output.

For input = -1 and output = 0:

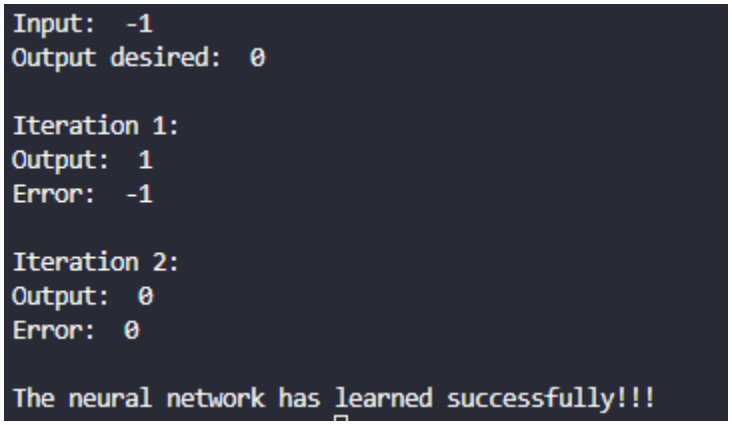

Figura 8 – Fifth example

Here, it suggests that the neuron should remain inactive when faced with negative input signals. The code works to adjust the neuron's weights so that it learns to suppress its output for negative inputs. If successful, the neuron will generate the desired output of zero when the input is negative.

For input = -1 and output = 1:

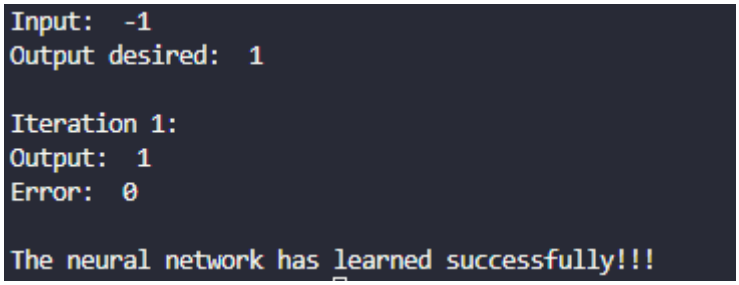

Figure 9 – Sixth and final example

In this case, it shows that the neuron still receives a negative input (-1), but now it's expected to produce a positive output (1). This scenario implies that the neuron should activate and generate an output in response to negative input signals. The code focuses on tweaking the neuron's weights to enable it to respond with a positive output when presented with negative inputs. If the training is effective, the neuron will output the desired positive signal when the input is negative.

Discussion

The primary goal of this personal project was to delve into the theory and practical implementation of neural networks, particularly focusing on the fundamentals of deep learning using a single neuron. By simulating the behavior of a neuron in Python, the project aimed to understand how neural networks adapt and correct errors during the learning process. This exploration stemmed from a curiosity about neural network concepts and their real-world applications, motivating a deeper understanding of their behavior and potential.

Conclusion

The project utilized Python programming to simulate a single neuron and train it to correct errors after one or more iterations. Although it was just one neuron and, in the actual technologies are billions and billions, they’re in a more complex way adjusted to minimize errors and to improve the performance in generating desired outputs. By closely analyzing the neuron's output and error metrics, insights were gained into the mechanisms underlying neural network learning and adaptation.