Module 1

Representation Power of MLPs

Representation Power of Perceptron Networks. How Perceptron Networks is different from single perceptron. How to perceptron networks use to solve functions which cannot be solved by perceptron?

A network of perceptrons, also known as a perceptron network or a multilayer perceptron (MLP), has a high representation power. The multilayer perceptron is a feedforward neural network that consists of multiple layers of interconnected perceptrons or artificial neurons.

Perceptron to Perceptron Networks

Representation boolean functions using Perceptron Network,

AND function

The AND function takes two binary inputs (0 or 1) and returns 1 only if both inputs are 1; otherwise, it returns 0.

We can implement the AND function using a single perceptron with two input weights and a bias.

The weights and bias can be set as follows:

- w1 = 1

- w2 = 1
- bias = -1.5

The activation function used by the perceptron can be a step function where any input greater than or equal to 0 results in an output of 1, and any negative input results in an output of 0.

Here's the implementation in Python:

```
def AND(x1, x2):
    w1 = 1
    w2 = 1
    bias = -1.5
    activation = w1*x1 + w2*x2 + bias
    if activation >= 0:
        return 1
    else:
        return 0
```

OR function

The OR function takes two binary inputs (0 or 1) and returns 1 if at least one input is 1; otherwise, it returns 0.

Similar to the AND function, we can implement the OR function using a single perceptron with two input weights and a bias.

The weights and bias can be set as follows:

- w1 = 1
- w2 = 1
- bias = -0.5

Again, we use a step function as the activation function.

The implementation in Python:

```
def OR(x1, x2):
    w1 = 1
    w2 = 1
    bias = -0.5
    activation = w1*x1 + w2*x2 + bias
    if activation >= 0:
        return 1
    else:
        return 0
```

NOT function

The NOT function takes a single binary input (0 or 1) and returns the opposite value. In other words, if the input is 0, it returns 1, and if the input is 1, it returns 0.

To implement the NOT function using a perceptron, we need a single input weight, a bias, and an appropriate activation function.

The weights and bias can be set as follows:

- w = -1
- bias = 0.5

The activation function can again be a step function.

Here's the implementation in Python:

```
def NOT(x):
    w = -1
    bias = 0.5
    activation = w*x + bias
    if activation >= 0:
        return 1
    else:
        return 0
```

Here things change…

XOR function

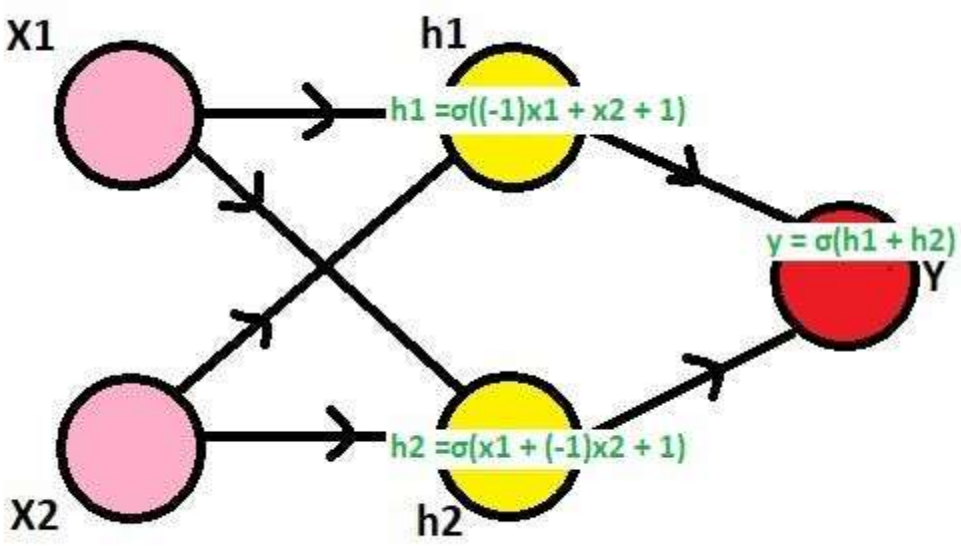
The XOR function takes two binary inputs (0 or 1) and returns 1 if the inputs are different (one is 0 and the other is 1); otherwise, it returns 0.

The XOR function cannot be implemented using a single perceptron since it is not a linearly separable function. However, we can achieve the XOR functionality by combining multiple perceptrons in a network.

A perceptron makes decisions based on a linear combination of its inputs, followed by applying an activation function. The decision boundary of a perceptron is a hyperplane, which is a straight line in two-dimensional space. However, the XOR function requires a non-linear decision boundary, which cannot be achieved by a single perceptron.

To implement the XOR function, we need a network of perceptrons, specifically a multilayer perceptron (MLP) with at least one hidden layer. The hidden layer(s) allow the network to learn non-linear mappings between the inputs and outputs. By combining multiple perceptrons and using non-linear activation functions, an MLP can model complex relationships and achieve the necessary non-linear decision boundaries to implement the XOR function.

Module 1



source

An example of a network with two layers to implement XOR using perceptrons:

```
# First layer perceptrons
def AND(x1, x2):
    # Implementation of the AND function as shown before

def NAND(x1, x2):
    # Implementing the NAND function using a perceptron
    w1 = -1
    w2 = -1
    bias = 1.5
    activation = w1*x1 + w2*x2 + bias
    if activation >= 0:
        return 1
    else:
        return 0

def OR(x1, x2):
    # Implementation of the OR function as shown before

# Second layer perceptron
def XOR(x1, x2):
    # Implementing XOR using the previous layer perceptrons
```

```
    first_layer_output = NAND(x1, x2)
    second_layer_output = OR(x1, x2)
    output = AND(first_layer_output, second_layer_output)
    return output
```

Now we see that Perceptron Networks can solve all boolean functions.

Representation Power of Perceptron Networks

The representation power of a network refers to its ability to approximate complex functions or learn intricate patterns from data. With a sufficient number of hidden layers and neurons, a multilayer perceptron can approximate any continuous function to arbitrary precision, given enough training data and appropriate training algorithms.

This property of multilayer perceptrons is known as the universal approximation theorem. It states that a feedforward neural network with a single hidden layer, containing a finite number of neurons, can approximate any continuous function on a closed and bounded input space. By increasing the number of neurons and adding more hidden layers, the representation power of the network increases, allowing it to capture more complex relationships in the data.

However, it is important to note that the universal approximation theorem does not provide any insights into the efficiency or ease of training such networks. The training process of multilayer perceptrons

can be challenging, particularly for deep networks with many layers, and it often requires careful initialization, regularization techniques, and optimization algorithms to achieve good results.

The key factor influencing the representation power of an MLP is the number of layers and neurons within each layer. As the number of layers and neurons increases, the network becomes capable of representing more complex relationships in the data.

The universal approximation theorem states that a multilayer perceptron with a single hidden layer, containing a finite number of neurons, can approximate any continuous function on a closed and bounded input space. This theorem guarantees that, theoretically, a single hidden layer MLP can capture a wide range of complex functions.

However, in practice, deep MLPs with multiple hidden layers tend to offer greater representation power. By adding more layers, the network can learn hierarchical representations, where each layer captures progressively more abstract and complex features from the input data. This allows deep MLPs to effectively model intricate patterns and relationships in the data.

The activation function used by the perceptrons in the network also plays a role in the representation power. Traditionally, perceptrons use step functions as activation functions, which limit the expressiveness of the

network. However, by using non-linear activation functions such as sigmoid, tanh, or ReLU, the network can model more complex and non-linear relationships between input and output.