Using the McCulloch-Pitts neuron

Using the McCulloch-Pitts neuron

To create the reward matrix, *R*, a robust model for processing the inputs of the huge volumes in a warehouse must be reduced to a limited number of features.

In one model, for example, the thousands to hundreds of thousands of inputs can be described as follows:

- Forecast product arrivals with a low priority weight: $w_1 = 10$
- Confirmed arrivals with a high priority weight: $w_2 = 70$
- Unplanned arrivals decided by the sales department: $w_3 = 75$
- Forecasts with a high priority weight: $w_4 = 60$
- Confirmed arrivals that have a low turnover and so have a low weight: $w_5 = 20$

The weights have been provided as constants. A McCulloch-Pitts neuron does not modify weights. A perceptron neuron does as we will see beginning with *Chapter 8*, *Solving the XOR Problem with a Feedforward Neural Network*. Experience shows that modifying weights is not always necessary.

These weights form a vector, as shown here:

$$x = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{bmatrix} = \begin{bmatrix} 10 \\ 70 \\ 75 \\ 60 \\ 20 \end{bmatrix}$$

Each element of the vector represents the weight of a feature of a product stored in optimal locations. The ultimate phase of this process will produce a reward matrix, *R*, for an MDP to optimize itineraries between warehouse locations.

Let's focus on our neuron. These weights, used through a system such as this one, can attain up to more than 50 weights and parameters per neuron. In this example, 5 weights are implemented. However, in real-life case, many other parameters come into consideration, such as unconfirmed arrivals, unconfirmed arrivals with a high priority, confirmed arrivals with a very low priority, arrivals from locations that probably do not meet security standards, arrivals with products that are potentially dangerous and require special care, and more. At that point, humans and even classical software cannot face such a variety of parameters.

The reward matrix will be size 6×6. It contains six locations, A to F. In this example, the six locations, `l1` to `l6`, are warehouse storage and retrieval locations.

A 6×6 reward matrix represents the target of the McCulloch-Pitts layer implemented for the six locations.

Using the McCulloch-Pitts neuron

When experimenting, the reward matrix, *R*, can be invented for testing purposes. In real-life implementations, you will have to find a way to build datasets from scratch. The reward matrix becomes the output of the preprocessing phase. The following source code shows the input of the reinforcement learning program used in the first chapter. The goal of this chapter describes how to produce the following reward matrix that we will be building in the next sections.

```
# R is The Reward Matrix for each location in a warehouse (or any other problem)
R = ql.matrix([ [0,0,0,0,1,0],
                [0,0,0,1,0,1],
                [0,0,100,1,0,0],
                [0,1,1,0,1,0],
                [1,0,0,1,0,0],
                [0,1,0,0,0,0] ])
```

For the warehouse example that we are using as for any other domain, the McCulloch-Pitts neuron sums up the weights of the input vector described previously to fill in the reward matrix.

Each location will require its neuron, with its weights.

INPUTS -> WEIGHTS -> BIAS -> VALUES

- Inputs are the flows in a warehouse or any form of data.
- Weights will be defined in this model.
- Bias is for stabilizing the weights. Bias does exactly what it means. It will tilt weights. It is very useful as a referee that will keep the weights on the right track.
- Values will be the output.

There are as many ways as you can imagine to create reward matrices. This chapter describes one way of doing it that works.

The McCulloch-Pitts neuron

The McCulloch-Pitts neuron dates back to 1943. It contains inputs, weights, and an activation function. Part of the preprocessing phase consists of selecting the right model. The McCulloch-Pitts neuron can represent a given location efficiently.

The following diagram shows the McCulloch-Pitts neuron model:

Using the McCulloch-Pitts neuron

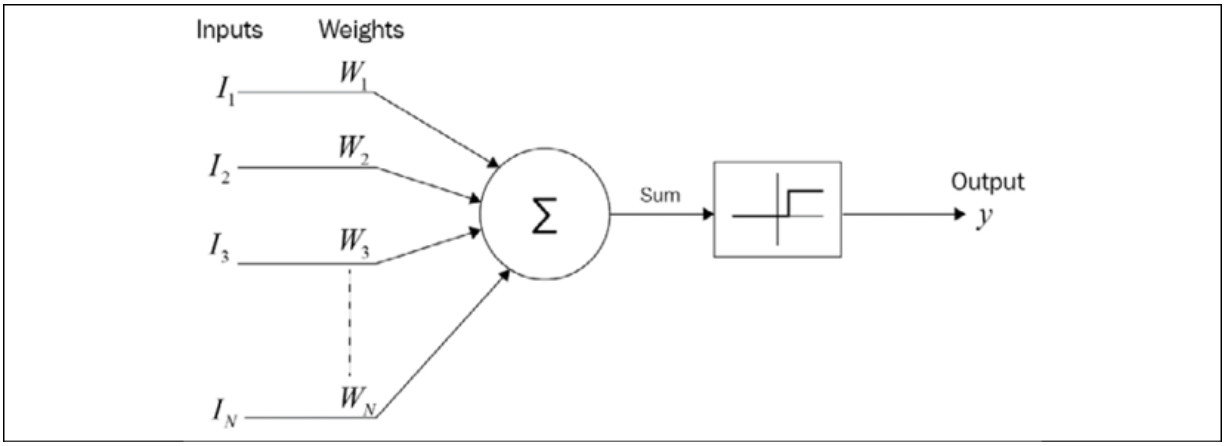

Figure 2.1: The McCulloch-Pitts neuron model

This model contains several input *x* weights that are summed to either reach a threshold that will lead, once transformed, to the output, *y* = 0, or 1. In this model, *y* will be calculated in a more complex way.

`MCP.py` written with TensorFlow 2 will be used to illustrate the neuron.

In the following source code, the TensorFlow variables will contain the input values (`x`), the weights (`W`), and the bias (`b`). Variables represent the structure of your graph:

```
# The variables
x = tf.Variable([[0.0,0.0,0.0,0.0,0.0]], dtype = tf.float32)
W = tf.Variable([[0.0],[0.0],[0.0],[0.0],[0.0]], dtype =
    tf.float32)
b = tf.Variable([[0.0]])
```

In the original McCulloch-Pitts artificial neuron, the inputs (*x*) were multiplied by the following weights:

$$w_1 x_1 + \cdots + w_n x_n = \sum_{j=1}^{n} w_j x_j$$

The mathematical function becomes a function with the neuron code triggering a logistic activation function (sigmoid), which will be explained in the second part of the chapter. Bias (`b`) has been added, which makes this neuron format useful even today, shown as follows:

```
# The Neuron
def neuron(x, W, b):
    y1=np.multiply(x,W)+b
    y1=np.sum(y1)
    y = 1 / (1 + np.exp(-y1))
    return y
```

Using the McCulloch-Pitts neuron

Before starting a session, the McCulloch-Pitts neuron (1943) needs an operator to set its weights. That is the main difference between the McCulloch-Pitts neuron and the perceptron (1957), which is the model for modern deep learning neurons. The perceptron optimizes its weights through optimizing processes. *Chapter 8*, *Solving the XOR Problem with a Feedforward Neural Network*, describes why a modern perceptron was required.

The weights are now provided, and so are the quantities for each input value, which are stored in the *x* vector at l_1, one of the six locations of this warehouse example:

$$x = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{bmatrix} = \begin{bmatrix} 10 \\ 70 \\ 75 \\ 60 \\ 20 \end{bmatrix}$$

The weight values will be divided by 100, to represent percentages in terms of 0 to 1 values of warehouse flows in a given location. The following code deals with the choice of *one* location, l_1 **only**, its values, and parameters:

```
# The data
x_1 = [[10, 2, 1., 6., 2.]]
w_t = [[.1, .7, .75, .60, .20]]
b_1 = [1.0]
```

The neuron function is called, and the weights (`w_t`) and the quantities (`x_1`) of the warehouse flow are entered. Bias is set to `1` in this model. No session needs to be initialized; the neuron function is called:

```
# Computing the value of the neuron
value=neuron(x_1,w_t,b_1)
```

The neuron function `neuron` will calculate the value of the neuron. The program returns the following value:

```
value for threshold calculation:0.99999
```

This value represents the activity of location l_1 at a given date and a given time. This example represents only one of the six locations to compute. For this location, the higher the value, the closer to 1, the higher the probable saturation rate of this area. This means there is little space left to store products at that location. That is why the reinforcement learning program for a warehouse is looking for the **least loaded** area for a given product in this model.

Each location has a probable **availability**:

A = Availability = 1 – load

The probability of a load of a given storage point lies between 0 and 1.

Using the McCulloch-Pitts neuron

High values of availability will be close to 1, and low probabilities will be close to 0, as shown in the following example:

```
>>> print("Availability of location x:{0:.5f}".format(
...       round(availability,5)))
Availability of location x:0.00001
```

For example, the load of l_1 has a probable rounded load of 0.99, and its probable *availability* is 0.002 maximum. The goal of the AGV is to search and find the closest and most available location to optimize its trajectories. l_1 is not a good candidate at that day and time. **Load** is a keyword in production or service activities. The less available resources have the highest load rate.

When all six locations' availabilities have been calculated by the McCulloch-Pitts neuron—each with its respective quantity inputs, weights, and bias—a location vector of the results of this system will be produced. Then, the program needs to be implemented to run all six locations and not just one location through a recursive use of the one neuron model:

$$A(L) = \{a(l_1),a(l_2),a(l_3),a(l_4),a(l_5),a(l_6)\}$$

The availability, 1 – *output value of the neuron*, constitutes a six-line vector. The following vector, l_v, will be obtained by running the previous sample code on **all** six locations.

$$l_v = \begin{bmatrix} 0.0002 \\ 0.2 \\ 0.9 \\ 0.0001 \\ 0.4 \\ 0.6 \end{bmatrix}$$

As shown in the preceding formula, l_v is the vector containing the value of each location for a given AGV to choose from. The values in the vector represent availability. 0.0002 means little availability; 0.9 means high availability. With this choice, the MDP reinforcement learning program will optimize the AGV's trajectory to get to this specific warehouse location.

The l_v is the result of the weighing function of six potential locations for the AGV. It is also a vector of transformed inputs.

References

Rothman. Denis, Artificial Intelligence By Example - Second Edition, PUBLISHED BY: Packt Publishing, PUBLICATION DATE:February 2020, available on http://go.oreilly.com/centennial-college.