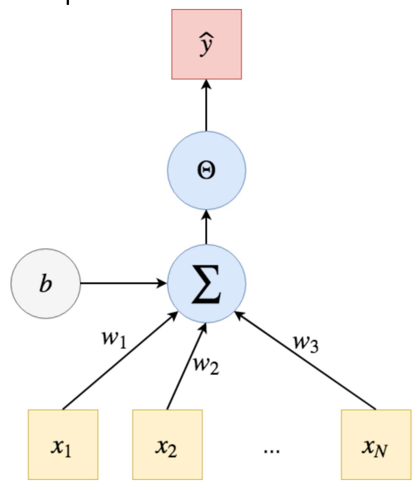
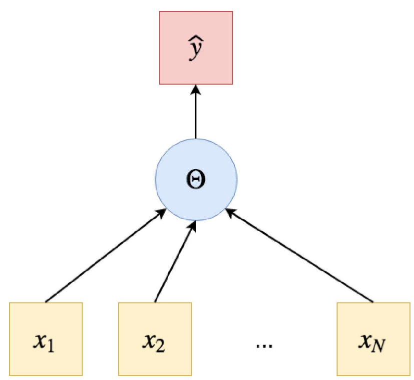
Perceptron algorithm for logic gates.

In [0]:
```
import numpy as np
```

The computational graph of our perceptron is:



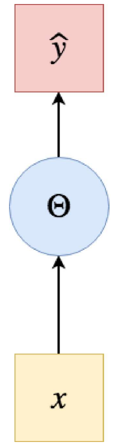
The Σ symbol represents the linear combination of the inputs x by means of the weights w and the bias b. Since this notation is quite heavy, from now on I will simplify the computational graph in the following way:



In [0]:
```
def unit_step(v):
    if v >= 0:
        return 1
    else:
        return 0
def perceptron(x, w, b):
    v = np.dot(w, x) + b
    y = unit_step(v)
    return y
```

NOT logical function

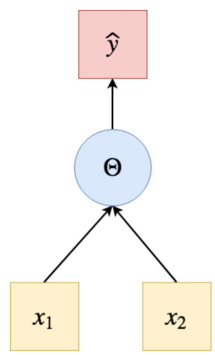
NOT(x) is a 1-variable function, that means that we will have one input at a time: N=1. Also, it is a logical function, and so both the input and the output have only two possible states: 0 and 1 (i.e., False and True): the Heaviside step function seems to fit our case since it produces a binary output.



The fundamental question is: do exist two values that, if picked as parameters, allow the perceptron to implement the NOT logical function? When I say that a perceptron implements a function, I mean that for each input in the function's domain the perceptron returns the same number (or vector) the function would return for the same input.

In [0]:
```
def NOT_percep(x):
    return perceptron(x, w=-1, b=0.5)
```

AND logical function



The AND logical function is a 2-variables function, AND(x1, x2), with binary inputs and output. This graph is associated with the following computation: ŷ = Θ(w1*x1* + *w2*x2 + b) This time, we have three parameters: w1, w2, and b. w1 = 1, w2 = 1, b = -1.5

In [0]:
```
def AND_percep(x1,x2):
    w = np.array([1, 1])
    b =-1.5
    x = np.array([x1,x2])
    return perceptron(x, w, b)
```

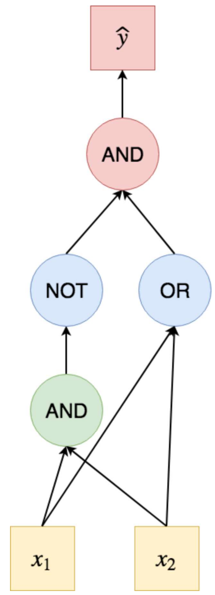
OR logical function

OR(x1, x2) is a 2-variables function too, and its output is 1-dimensional (i.e., one number) and has two possible states (0 or 1). Therefore, we will use a perceptron with the same architecture as the one before. w1 = 1, w2 = 1, b = -0.5

In [0]:
```
def OR_percep(x1,x2):
    w = np.array([1, 1])
    b =-0.5
    x = np.array([x1,x2])
    return perceptron(x, w, b)
```

XOR logical function

$$XOR(x1, x2) = AND(NOT(AND(x1, x2)), OR(x1, x2))$$



In [0]:
```
def XOR_net(x1,x2):
    out_1 = AND_percep(x1,x2)
    out_2 = NOT_percep(out_1)
    out_3 = OR_percep(x1,x2)
    output = AND_percep(out_2,out_3)
    return output
```

In [0]:
```
print("NOT(0) = {}".format(NOT_percep(0)))
print("NOT(1) = {}".format(NOT_percep(1)))
```

```
NOT(0) = 1
NOT(1) = 0
```

In [24]:
```
print("AND({}, {}) = {}".format(1, 1, AND_percep(1,1)))
print("AND({}, {}) = {}".format(1, 0, AND_percep(1,0)))
print("AND({}, {}) = {}".format(0, 1, AND_percep(0,1)))
print("AND({}, {}) = {}".format(0, 0, AND_percep(0,0)))
```

```
AND(1, 1) = 1
AND(1, 0) = 0
AND(0, 1) = 0
AND(0, 0) = 0
```

In [25]:
```
print("OR({}, {}) = {}".format(1, 1, OR_percep(1,1)))
print("OR({}, {}) = {}".format(1, 0, OR_percep(1,0)))
print("OR({}, {}) = {}".format(0, 1, OR_percep(0,1)))
print("OR({}, {}) = {}".format(0, 0, OR_percep(0,0)))
```

```
OR(1, 1) = 1
OR(1, 0) = 1
OR(0, 1) = 1
OR(0, 0) = 0
```

In [26]:
```
print("OR({}, {}) = {}".format(1, 1, XOR_net(1,1)))
print("OR({}, {}) = {}".format(1, 0, XOR_net(1,0)))
print("OR({}, {}) = {}".format(0, 1, XOR_net(0,1)))
print("OR({}, {}) = {}".format(0, 0, XOR_net(0,0)))
```

```
OR(1, 1) = 0
OR(1, 0) = 1
OR(0, 1) = 1
OR(0, 0) = 0
```

In [0]:
```

```