# Uso de um Perceptron para aprender o funcionamento de portas lo?gicas

April 8, 2018

## 1 Uso de um Perceptron para aprender o funcionamento de portas lógicas

```
In [1]: import math
        import random
        import matplotlib.pylab as plt
```

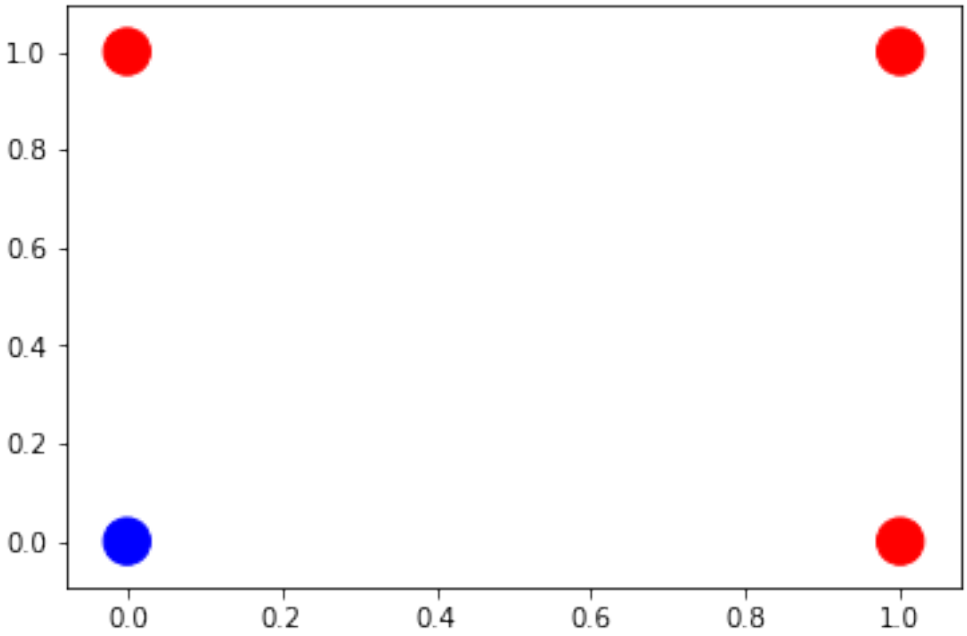
Para este tutorial, vamos treinar um perceptron para aprender a calcular corretamente as saídas de uma porta lógica. Aqui, usaremos a porta OU. Como a saída da porta é binária, pode-se tratar o problema como sendo de classificação. Assim, apenas duas saídas são possíveis: zero ou um.

Inicialmente, devemos construir o conjunto de entradas e o de saídas.

```
In [2]: entradas = [[0,0], [0,1], [1,0], [1,1]]
        saidas = [0, 1, 1, 1]
```

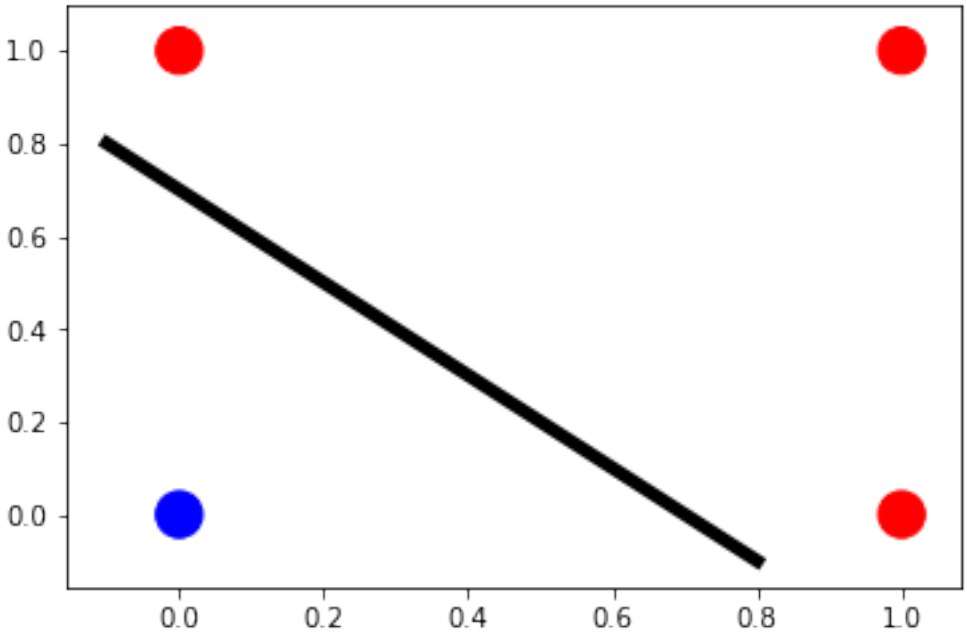
Vamos fazer o plot para verificar como esses pontos se posicionam no espaço.

```
In [3]: plt.figure()
        x = [e[0] for e in entradas]
        y = [e[1] for e in entradas]
        cores = [ ['b', 'r'][idx] for idx in saidas  ]
        plt.scatter(x, y, color=cores, s=300)
        plt.show()
```




Como se pode perceber, existe uma separação linear entre os valores. Isso significa que é possível separar o ponto azul dos vermelhos com uma reta. Por exemplo, essa reta pode passar nos eixos x=0,8 e y=0,8. Com isso, os valores acima da reta são classificados como vermelho (1) e os abaixo como azul (0).

```
In [4]: plt.figure()
        x = [e[0] for e in entradas]
        y = [e[1] for e in entradas]
        cores = [ ['b', 'r'][idx] for idx in saidas  ]
        plt.scatter(x, y, color=cores, s=300)
        plt.plot([0.8, -0.1], [-0.1, 0.8], 'k-', linewidth=5)
        plt.show()
```




Porém, existem infinitas retas que podem ser traçadas. O algoritmo de treinamento do Perceptron deve ser capaz de encontrar uma delas para resolver o problema.

O Perceptron funciona como a soma ponderada de suas entradas, resultando no potencial de ativação $u$:

$$u = \sum_{i=0}^{n} p_i \cdot x_i$$

em que $\overrightarrow{p}$ é o vetor de pesos, $\overrightarrow{x}$ é o vetor de entradas de um único exemplo e $n$ é o tamanho de $\overrightarrow{x}$. A seguir, aplica-se uma função de transferência (função de ativação) nesse potencial para se obter a saída final. Essa função de ativação pode fazer com que o Perceptron seja um modelo não-linear, visto que os pesos ficariam *dentro* da função de ativação. Para o problema estudado aqui, empregaremos a função degrau para converter o valor real em binário:

$$degrau(u) = \begin{cases} 1 & \text{se } u \geq 0 \\ 0 & \text{caso contrário} \end{cases}$$

```
In [5]: degrau = lambda u: 1 if u >= 0 else 0
```

Como a reta não precisa passar obrigatoriamente pela origem (0, 0), precisamos adicionar um peso extra que é o coeficiente linear conhecido como *viés* (*bias*). Assim, no cálculo de $s$, deve-se incluir um elemento neutro de valor 1 para multiplicar pelo *viés*. Com isso, o vetor de pesos tem um valor a mais.

O ajuste dos pesos é chamado *treinamento*. Esse ajuste é feito no perceptron usando a regra delta:

$$p_{i+1} = p_i + alfa * erro * x_i$$

em que alfa é a taxa de aprendizado que indica o grau de ajuste do peso em direção à entrada. Note que cada peso pode ter um ajuste distinto visto que se considera o erro da saída e o valor da entrada no mesmo índice do peso.

De acordo com a teoria, uma taxa de aprendizado alta faz com que o ajuste do peso seja mais rápido. Contudo, pode ocorrer rapidamente o problema de sobreajuste fazendo com que a reta se aproxime demasiadamente dos exemplos, reduzindo a capacidade de generalização.

O processo de treinamento ocorre em várias épocas, sendo que uma época corresponde à apresentação de todos os dados de entrada e o ajuste dos pesos para cada entrada. Caso nenhum erro seja verificado, ou o erro acumulado seja menor do que uma determinada *tolerância*, pode-se finalizar o treinamento. Para isso, emprega-se um acumulador de erros (soma_erros).

A seguir, vamos fazer um único passo, apresentando um único exemplo, calculando a soma e saída e ajustando os pesos. Inicialmente, é necessário inicializar os pesos com valores arbitrários.

```
In [6]: pesos = [0.1, 0.3, -1.0]

        alfa = 0.5
        soma_erros = 0
        tolerancia = 1e-4
        idx = 0 # primeiro exemplo
        x = [1] + entradas[idx]  # inclusao do bias
        potencial = sum ([pesos[i] * x[i] for i in range(len(x))])
        saida_calculada = degrau(potencial)

        print ("entradas[idx] =", entradas[idx])
        print ("x =", x)
        print ("potencial =", potencial)
        print ("saida_calculada =", saida_calculada)
```

```
entradas[idx] = [0, 0]
x = [1, 0, 0]
potencial = 0.1
saida_calculada = 1
```

O erro é a diferença entre a saída esperada e a saída calculada.

```
In [7]: erro = saidas[idx] - saida_calculada
        soma_erros += abs(erro)

        print ("erro =", erro)
        print ("soma_erros =", soma_erros)
```

```
erro = -1
soma_erros = 1
```

Ajusta-se os pesos de acordo com o erro calculado:

```
In [8]: print ("pesos antes da atualizacao =", pesos)

        pesos = [pesos[i] + (alfa * erro * x[i]) for i in range(len(x))]

        print ("pesos apos a atualizacao =", pesos)
```

```
pesos antes da atualizacao = [0.1, 0.3, -1.0]
pesos apos a atualizacao = [-0.4, 0.3, -1.0]
```

Finalizada uma época, ou seja, a apresentação de todas as entradas, verifica-se o critério de parada do treinamento:

```
In [9]: if soma_erros < tolerancia: break
```

```
  File "<ipython-input-9-372a09bdfc9e>", line 1
    if soma_erros < tolerancia: break
                                ^
SyntaxError: 'break' outside loop
```

Obviamente, o código acima está incompleto, pois não existem o laço de apresentação dos exemplos nem o de épocas de treinamento. Fica, como exercício, obter um código completo e funcional.