Notebook

March 30, 2018

1 Lista de Exercícios 1

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

1.1 Parte 1 - Exercícios conceituais

1.1.1 1. Explique com suas palavras o que é um neurônio artificial e o seu funcionamento traçando uma analogia com o neurônio biológico.

O neurônio artificial é uma unidade simples de processamento, capaz de fazer um cálculo matemático com base em várias entradas e gerar uma saída, a qual pode ser conectada à entrada de outros neurônios artificiais. A conjunção de várias dessas unidades de processamento na forma de rede possibilita a execução de cálculos mais complexos. Toda essa ideia foi inspirada no funcionamento do neurônio biológico. Esse também pode ser visto como uma unidade simples de processamento, o qual recebe múltiplos sinais eléstricos e gera uma saída, a qual é ligada a outros neurônios. Apesar de ser uma unidade tão simples, a grande quantidade existente no cérebro é o que gera toda sua capacidade.

1.1.2 2. O que são e para que servem os notebooks de programação?

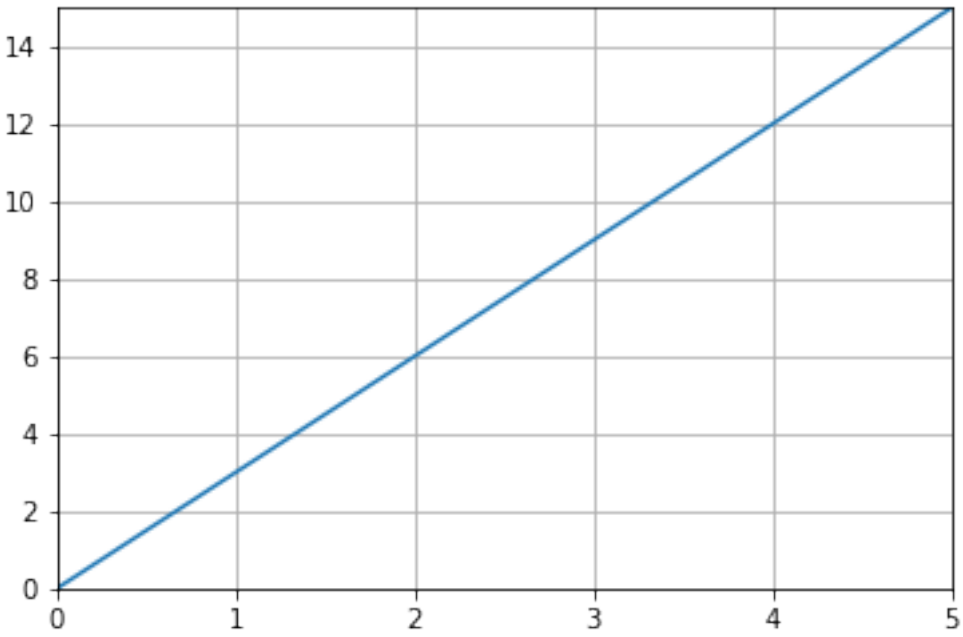
Notebooks são ferramentas de programação que permite: 1) Geração de conteúdo através de texto e código (em diversas linguages através dos kernels). 2) Executar código de maneira interativa, visualizando diretamente o resultado, seja ele uma simples resposta da chamada de um método até a exibição de gráficos e figuras. Eles são comumente utilizados para prototipação, visto que os resultados podem ser visualizados imediamente, permitindo fácil alteração das células de código e sua reexecução. Também é utilizado para facilitar a reprodutibilidade de material científico, visto que determinado artigo feito em um Notebook já mostra o código utilizado e facilita que o leitor o execute novamente. Em alguns momentos, também é utilizado para ensino, visto que o aluno também pode acessar facilmente o código utilizado.

1.1.3 3. Explique para que serve o Bias.

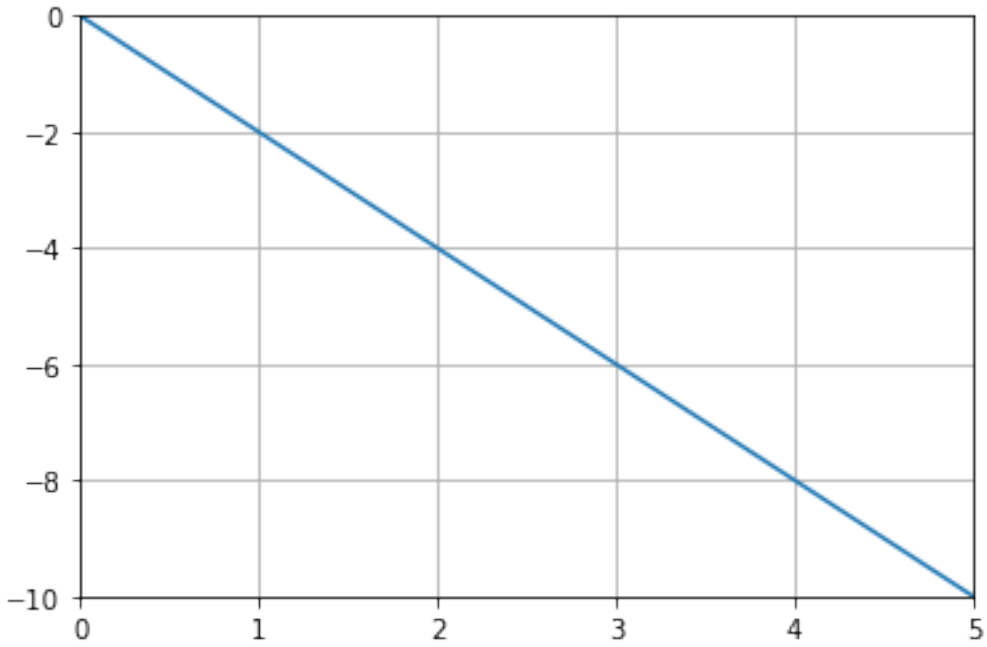
O Bias serve para representar um viés do problema. Matematicamente, ele permite ajustar a reta ou curva gerada pela função matemática de saída do neurônio. Sem ela, um classificador, por exemplo, só conseguiria gerar retas saindo do ponto zero. Para exemplificar, os gráficos abaixo mostram equações com e sem um Bias.

In [32]:
```
def plot_linear_function(func, y_lim):
    x = np.array([0, 5])
    y = func(x)
    plt.grid()
    plt.margins(x=0)
    plt.ylim(y_lim)
    plt.plot(x, y)
    plt.show()
```

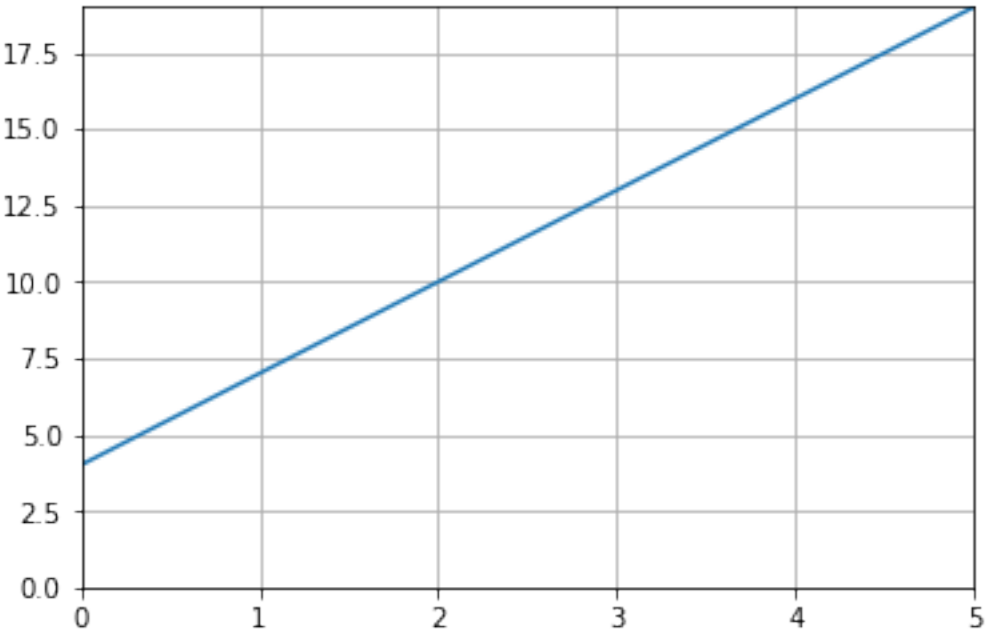
In [33]:
```
# 3x
plot_linear_function(lambda x: 3*x, [0, 15])
```



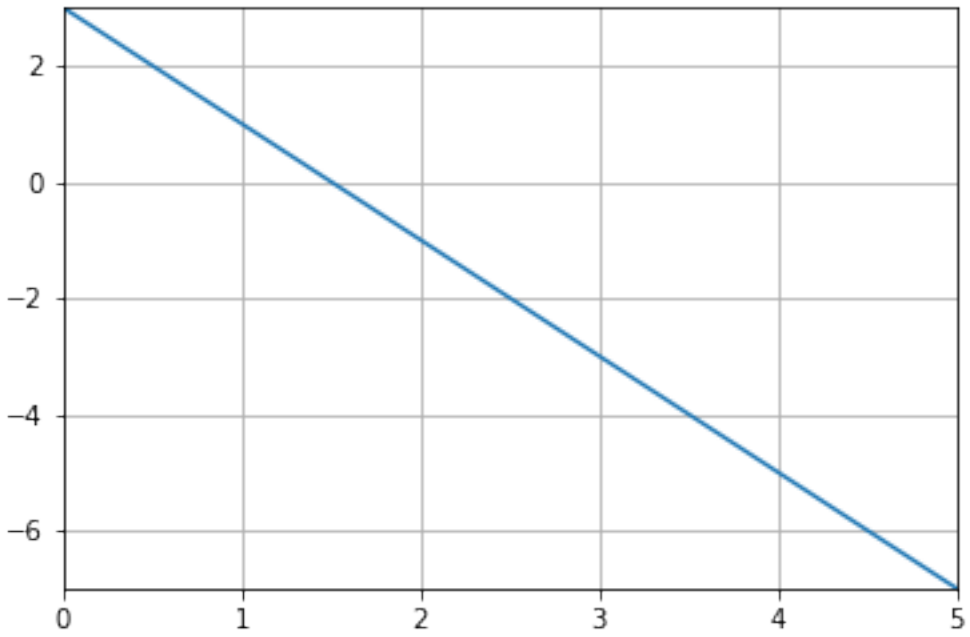
In [35]:
```
# -2x
plot_linear_function(lambda x: -2*x, [-10, 0])
```



In [39]:
```
# 3x + 4
plot_linear_function(lambda x: 3*x + 4, [0, 19])
```



```
In [40]: # -2x + 3
         plot_linear_function(lambda x: -2*x + 3, [-7, 3])
```

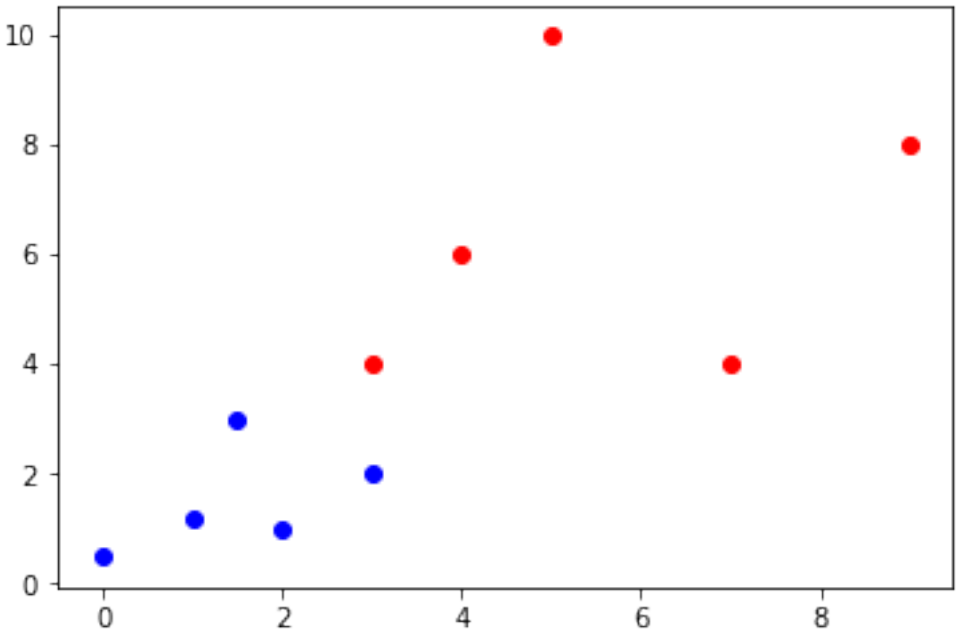


Como pode-se notar pelos gráficos, sem o Bias (o termo que não é multiplicado por uma das entradas), todas as retas têm a origem no zero. Com retas assim, não seria possível separar nem mesmo problemas linearmente separáveis.

1.1.4 4. Explique matematicamente em um plano o efeito do uso do Bias na definição de um hiperplano para um problema hipotético.

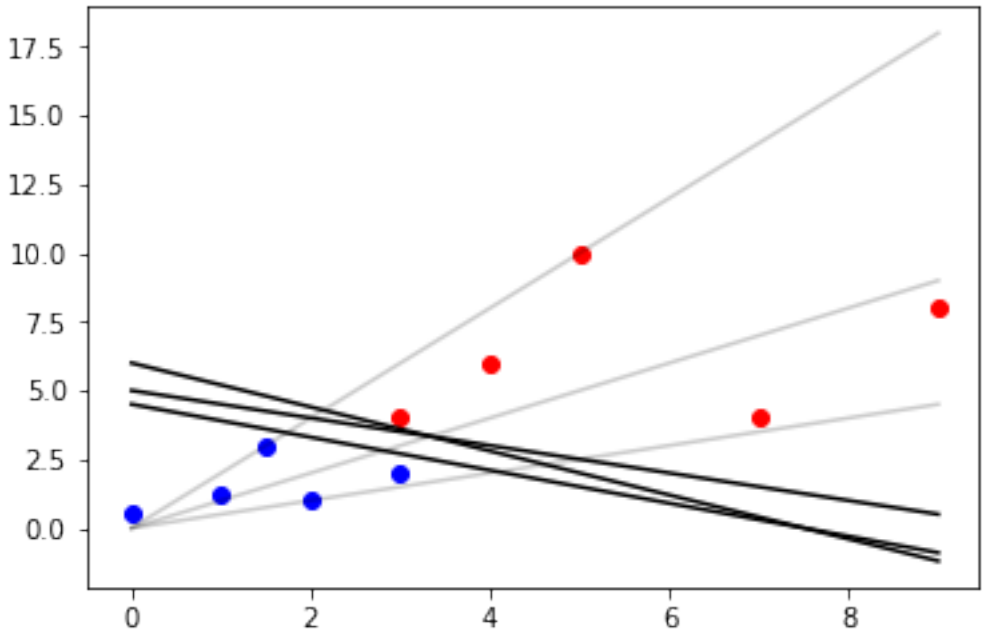
O bias tem o efeito de transladar um hiperplano, tornando possível que esse separe elementos em duas categorias. Para exemplificar, o gráfico a seguir mostra elementos de duas categorias diferentes:

```
In [41]: plt.scatter([3, 4, 7, 5, 9], [4, 6, 4, 10, 8], color='red')
         plt.scatter([1, 0, 2, 1.5, 3], [1.2, 0.5, 1, 3, 2], color='blue')
         plt.show()
```



Existem diversos hiperplanos capazes de separar essas duas categorias, mas todos eles necessitam do bias, como pode ser visto a seguir:

```
In [55]: def plot_hyperplane(func, alpha=1):
             x = np.array([0, 9])
             y = func(x)
             plt.plot(x, y, alpha=alpha, color='black')

         plt.scatter([3, 4, 7, 5, 9], [4, 6, 4, 10, 8], color='red')
         plt.scatter([1, 0, 2, 1.5, 3], [1.2, 0.5, 1, 3, 2], color='blue')
         plot_hyperplane(lambda x: 2*x, 0.2)
         plot_hyperplane(lambda x: 0.5*x, 0.2)
         plot_hyperplane(lambda x: x, 0.2)
         plot_hyperplane(lambda x: -0.5*x+5)
         plot_hyperplane(lambda x: -0.6*x+4.5)
         plot_hyperplane(lambda x: -0.8*x+6)
         plt.show()
```



1.1.5 5. Explique conceitualmente o funcionamento do tensorflow estabelecendo uma relação entre nós e arestas de um grafo.

O Tensorflow permite a configuração prévia de uma série de operações matemáticas a serem executadas no futuro. Para isso, ele representa essas operações em uma estrutura de DAG (Directed acyclic graph). Ou seja, cada operação é representado por um nó e suas dependências são representadas por arestas que conectam esses nós. Se uma operação matemática depende do resultado de duas outras, essas serão executadas primeiro e seus resultados serão utilizados na operação seguinte. Esse conceito pode ser utilizado tanto por operações mais simples, como uma equação, como também operações muito mais complexas, as quais são abstrações de operações mais simples.

1.1.6 6. Quais as principais diferenças no fluxo de execução no tensorflow em relação a estruturas de dados convencionais?

Em estruturas convencionais, a execução acontece no momento em que cada linha de código é executada. Assim, é comum armazenar os valores em variáveis e utilizá-las em outras computações. No Tensorflow, existem dois momentos: o momento em que o grafo de execução é configurado e o momento em que uma sessão é aberta e esse grafo é executado pelo Tensorflow. A vantagem dessa abordagem é que o Tensorflow pode preparar um plano de como irá executar esse grafo de operações, podendo inclusive distribuí-las em múltiplos núcleos de processamento e até mesmo executá-las em GPU(s) com instruções de CUDA. Sem o uso do Tensorflow, teríamos que implementar manualmente essa distribuição, o que é bem complexo, especialmente se for necessário utilizar código em CUDA. Além disso, é possível armazenar esse grafo e executá-lo depois. Também é possível criar placeholder onde serão inseridos os dados de entrada durante a execução.

1.1.7 7. Descreva o objetivo do uso da função de ativação de um neurônio.

O objetivo da função de ativação de um neurônio é adicionar não linearidade em sua resposta. Do contrário, mesmo com redes de múltiplas camadas, só seria possível criar retas através da somatória das entradas.

1.1.8 8. Qual o objetivo do uso de pesos nos neurônios artificiais? Dica: Faça uma analogia entre sistemas biológicos e artificiais.

O objetivo é ponderar a importância de cada entrada e indicar se ela terá um efeito positivo ou negativo na resposta final. Em sistemas biológicos, é como nosso cérebro consegue se concentrar em partes importantes da informação para chegar a um entendimento. O sistema que processa as imagens capturadas pelos olhos, por exemplo, consegue ponderar as partes mais importantes (como bordas, cores, etc.), criando abstrações para os próximos neurônios. Em sistemas artificiais, é como se decide se determinada característica é mais importante que outra na classificação ou regressão, por exemplo.

1.1.9 9. Um neurônio artificial que usa uma função de ativação não-linear é capaz de tratar problemas não-lineares? Justifique sua resposta.

Sim. Se duas classes forem separáveis por **uma** curva, ao invés de uma reta, é possível que a separação seja feita por um único neurônio com a função de ativação correta. Entretanto, é comum a combinação de múltiplos neurônios em rede para que vários hiperplanos possam ser gerados, tornando mais precisa a classificação.

1.1.10 10. Se um neurônio artificial do tipo perceptron for inicializado com todos os pesos iguais a zero ele irá convergir para uma solução aceitável no problema?

Sim. Dado o algoritmo de treinamento do Perceptron, ele conseguirá convergir em problemas linearmente separáveis. Isso porque ele utiliza um algoritmo simples que faz ajuste online dos pesos a cada erro, não fazendo uso de gradiente descendente ou técnicas do tipo.

1.1.11 11. Um neurônio artificial após o seu treinamento apresenta todos os pesos iguais. Como você interpretaria esse resultado?

Considerando que não houve algum erro no treinamento, significa que ele identificou que todas as features têm igual relevância, todas contribuindo no mesmo sentido (positivo ou negativo) para a resposta final.

1.1.12 12. Em caso do NÃO-USO do bias em um neurônio artificial o resultado do treinamento será o mesmo?

Não. Em alguns casos, nem será possível chegar em uma solução aceitável.

1.1.13 13. Para que serve a taxa de aprendizado no processo de treinamento de um neurônio artificial?

Ela serve para calibrar a alteração de pesos em cada passo do treinamento. Uma taxa de aprendizado alta faz com que as alterações aconteçam mais rápido, mas pode fazer com que os pesos

ultrapassem os valores corretos da solução. Em casos extremos, pode fazer com que a resposta piore a cada passo por passar direto da solução e se afastar muito. Enquanto isso, uma taxa de aprendizado baixa faz com que as alterações sejam muito lentas, necessitando uma quantidade maior de passos e/ou épocas. Além do tempo computacional extra, também pode fazer com que o algoritmo fique tenha dificuldades em sair de ótimos locais.

1.1.14 14. O que é e para que serve o gradiente?

a) Exemplifique conceitualmente. Dica: Exponha sua resposta para uma criança de 10 anos e verifique se ela entendeu. O gradiente é como se fosse uma bússula que aponta para o ponto mais alto. Se estivéssemos em uma área de serra, ele sempre apontaria para uma direção de subida imediata. Porém, além de apontar, ele também mostra a intensidade dessa subida. Se estivermos diante de uma subida muito íngrime, ele aponta para ela com um valor alto. Se estivermos num lugar quase plano, seu valor é muito menor. A utilidade é encontrar pontos mais altos ou mais baixos se seguirmos essa "bússola" no sentido em que ela aponta ou no sentido oposto.

b) Exemplifique matematicamente O gradiente indica a variação imediata de valor de uma função de acordo com determinada variável de entrada. Dada uma função, sua derivada apontará para a direção de maior valor dessa função com determinada intensidade. Para isso, basta derivarmos a função em relação a cada uma de suas entradas, obtendo esses valores e utilizando-os como um vetor. Esse vetor terá direção e sentido apontando para o crescimento do valor dessa função. Dessa forma, pode-se utilizá-lo para buscar por valores máximos ou mínimos dessa função.

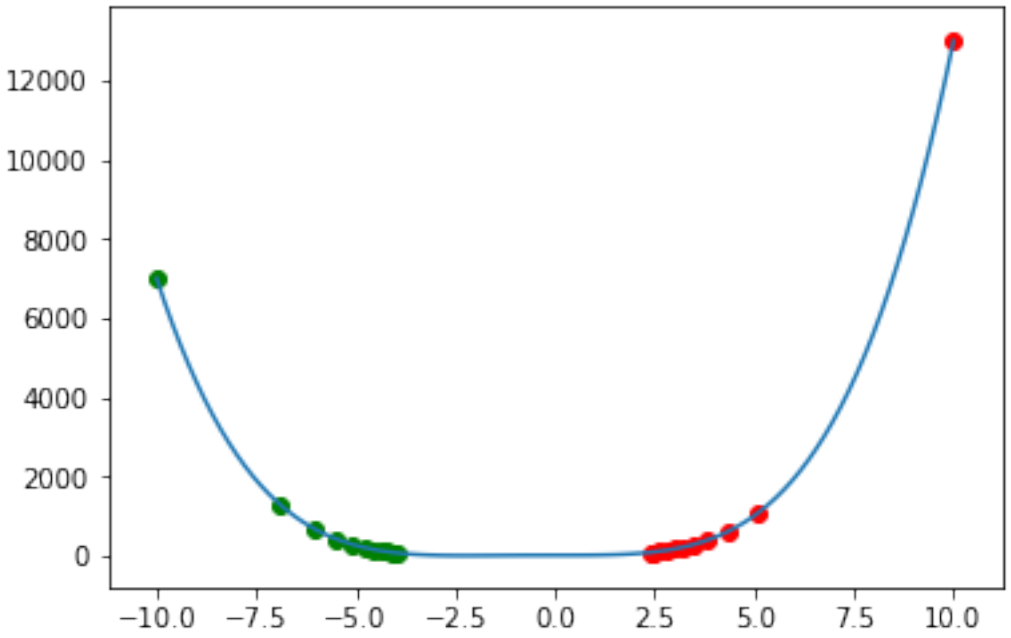
1.1.15 15. Sempre que a derivada de uma função é nula podemos afirmar que a função passa por um máximo ou mínimo?

Não exatamente. Se estiver em uma parte plana dessa função, o valor da derivada também será nula. Isso pode ser observado na segunda figura que demonstra um ponto de inflexão.

1.1.16 16. Dado a função y = f(x) = xˆ4 3xˆ3 + 2 escolha um valor aleatório de x e mostre com um exemplo numérico como o cálculo da derivada leva ao ponto de mínimo da função com uma determinada taxa de aprendizado.

```
In [61]: theta = 0.001

         def func(x):
             return x**4 + 3*x**3 + 2

         def derivative(x):
             return 4*x**3 + 9*x**2

         x = np.linspace(-10, 10, 100)
         plt.plot(x, func(x))

         def search(initial_value, color):
             x_values = [initial_value]

             for i in range(10):
```

```
            xt = x_values[-1]
            xt1 = xt - theta * derivative(xt)
            x_values.append(xt1)

        plt.scatter(x_values, func(np.array(x_values)), color=color)

search(10, 'red')
search(-10, 'green')
plt.show()
```



1.1.17 17. Qual a diferença entre o algoritmo de treinamento do neurônio perceptron e o Adaline?

Primeiramente, o Perceptron utiliza o **step** como função de ativação, convertendo os valores para 0 ou 1. Dessa forma, não é possível verificar o erro verdadeiro do neurônio, mas sim se ele gerou a resposta final correta. O treinamento do Perceptron simplesmente ajusta todos os pesos multiplicando a entrada pela subtração entre o valor esperado e o valor de saída. Dessa forma, ele tenta localizar os pesos "culpados" do erro e ajustá-los. Todo esse treinamento é feito online, ou seja, a cada passo do treinamento os pesos são ajustados.

Por outro lado, o Adaline utiliza uma função de ativação linear. Com isso, ela não verifica apenas se a resposta final está correta ou errada, mas também verifica o quanto ela se aproxima ou se afasta da resposta esperada. O treinamento do Adaline é baseado no gradiente descendente e é executado em batch. A cada época de treinamento, todos os erros são calculados e obtém-se a média quadrática do erro. Por fim, utiliza-se do gradiente para minimizar esse erro quadrático, ajustando os pesos a cada época.

A grande diferença entre os dois é que, dado uma taxa de aprendizagem suficientemente pequena, o adaline garante que a melhor reta que separa duas classes será encontrada. Enquanto o Perceptron pode gerar infinitas retas, variando de acordo com os valores iniciais dos pesos.

1.1.18 18. Independente dos valores inicias assumidos para o vetor de pesos do Adaline, uma mesma configuração final para w (final) será sempre obtida após a sua convergência. Essa afirmação é falsa ou verdadeira? Justifique.

Como explicado na questão anterior, essa afirmação é verdadeira. Isso porque o Adaline minimiza os erros quadráticos, ao invés de apenas corrigir o resultado final, como o Perceptron.

1.1.19 19. Explique a figura abaixo. (PDF)

Essa figura representa um hiperplano de decisão que poderia ser utilizado por um classificador. Ela tém essa forma de acordo com os pesos adotados pelo Perceptron com o fim do treinamento.

1.2 Parte 2 - Exercícios de Implementação

1.2.1 20. Considere o código do arquivo exercicio14.py. Construa um notebook e comente o que cada trecho de código indicado está realizando.

```
In [ ]: # Importando as bibliotecas necessárias
        # Numpy para processamento matemático de vetores e matrizes
        # Tensorflow para processamento matemático de Grafos, geralmente utilizado em Deep Learn
        import numpy as np
        import tensorflow as tf

        # Criação da sessão do Tensorflow
        sess = tf.Session()

        # Função criada para imprimir variáveis
        # Abaixo, uma constante é criada no grafo
        def print_tf(x):
            print("TIPO: \n %s" % (type(x)))
            print("Valor: \n %s" % (x))
        hello = tf.constant("www.deeplearningbrasil.com.br")
        print_tf(hello)

        # Utiliza a sessão do Tensorflow para executar o único nó presente no grafo até o moment
        # Depois, o resultado é imprimido na tela utilizando a função criada anteriormente
        hello_out = sess.run(hello)
        print_tf(hello_out)

        # Duas constantes são criadas e imprimidas
        # Porém, por não terem sido executadas pelo tensorflow ainda, ainda não são consideradas
        a = tf.constant(1.5)
        b = tf.constant(2.5)
        print_tf(a)
        print_tf(b)
```

```
# Os dois nós são executados separadamente, retornando o valor da constante (float)
# Então, o valor é imprimido
a_out = sess.run(a)
b_out = sess.run(b)
print_tf(a_out)
print_tf(b_out)

# Um novo nó é criado, representando a operação matemática entre a constante a e b
# Esse nó é imprimido na tela. Lembrando que ainda não é o valor da somatória
a_plus_b = tf.add(a, b)
print_tf(a_plus_b)

# A somatória é executada pelo Tensorflow e o valor é de 4 é imprimido na tela
a_plus_b_out = sess.run(a_plus_b)
print_tf(a_plus_b_out)

# Um novo nó representando a multiplicação da constante a e b é criado.
# Depois, ele é executado pelo tensorflow e o valor de 3,75 é imprimido
a_mul_b = tf.mul(a, b)
a_mul_b_out = sess.run(a_mul_b)
print_tf(a_mul_b_out)

# Cria um nó que representa uma variável que será inicializada futuramente.
# Essa variável receberá (quando for inicializada) uma matriz de dimensões 5x2 com valor
# vindos de uma distribuição normal de desvio padrão de 0.1.
weight = tf.Variable(tf.random_normal([5, 2], stddev=0.1))
print_tf(weight)

# Variáveis devem ser inicializadas antes que seus nós possam rodar no Tensorflow
weight_out = sess.run(weight)
print_tf(weight_out)

# Cria um nó de inicialização de todas variáveis presentes no grafo e o executa
init = tf.initialize_all_variables()
sess.run(init)

# Executa o nó da variável e printa o resultado
weight_out = sess.run(weight)
print_tf(weight_out)
print ("INITIALIZING ALL VARIALBES")

# Cria um placeholder, o qual poderá receber um valor no momento em que for rodar a sess
# do feed_dict. Ele possuirá uma quantidade indefinida de linhas e 5 colunas.
x = tf.placeholder(tf.float32, [None, 5])
print_tf(x)

# Cria uma nó no grafo representando a multiplicação da matriz presente na variável weig
```

```
# a ser populado.
oper = tf.matmul(x, weight)
print_tf(oper)

# Cria uma matriz com números aleatórios de dimensão 1x5
# Executa o nó da multiplicação das matrizes, colocando a matriz recém criada no placeho
# Depois disso, o resultado é printado na tela
data = np.random.rand(1, 5)
oper_out = sess.run(oper, feed_dict={x: data})
print_tf(oper_out)

# Executa a mesma operação, mas dessa vez com uma matriz de dimensão 2x5
data = np.random.rand(2, 5)
oper_out = sess.run(oper, feed_dict={x: data})
print_tf(oper_out)
```

1.2.2 21. Faça a implementação do algoritmo de treinamento do neurônio perceptron. Realize o treinamento do neurônio para a solução do problema do operador lógico OU partindo da inicialização do pesos abaixo:

w1: 0.3092

w2: 0.3129

bias: -0.8649

```
In [187]: X = np.array([[0, 0],
                        [0, 1],
                        [1, 0],
                        [1, 1]])
          y = np.array([0, 1, 1, 1])
```

```
In [307]: class Perceptron:

              def __init__(self, initial_weights):
                  self._weights = np.array(initial_weights)

              def _prepend_bias(self, X):
                  return np.insert(np.atleast_2d(X), 0, values=1, axis=1)

              def predict(self, X):
                  output = np.matmul(self._prepend_bias(X), self._weights)
                  return (output >= 0).astype(int)

              def fit(self, X, y):
                  n = X.shape[0]
                  weights_history = [self._weights]
                  errors_history = []
```

```
                for i in range (100):
                    for x, d in zip(X, y):
                        prediction = self.predict(x)
                        error = d-prediction[0]
                        delta = np.insert(x, 0, 1)*error
                        self._weights = self._weights + delta
                        weights_history.append(self._weights)
                        errors_history.append(error)
                        if len(errors_history) >= n and not np.any(errors_history[-n:]):
                            # Early stopping
                            return weights_history, errors_history

                return weights_history, errors_history
```

```
In [308]: perceptron = Perceptron([-0.8649, 0.3092, 0.3129])

In [309]: weights_history, errors_history = perceptron.fit(X, y)

In [310]: perceptron.predict(X)

Out[310]: array([0, 1, 1, 1])

In [311]: errors_history

Out[311]: [0, 1, 0, 0, -1, 0, 1, 0, -1, 0, 0, 0, 0]
```

a) O treinamento convergiu depois de 8 passos.

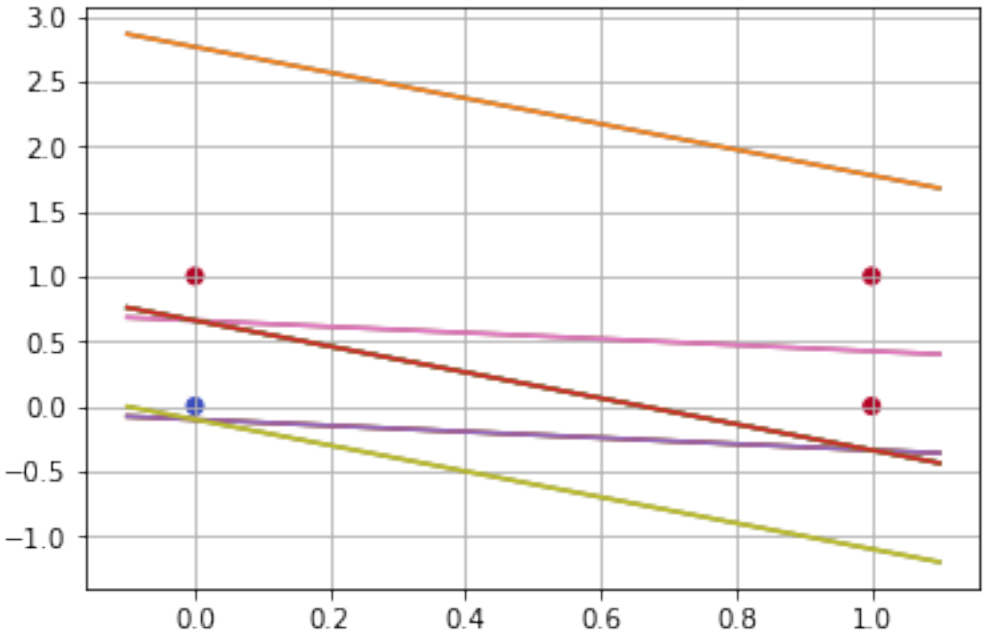
```
In [312]: def plot_or(weights):
              plt.grid()
              plt.scatter(X[:,0], X[:,1], c=y, cmap='coolwarm')

              for w in weights:
                  slope = -(w[0]/w[2])/(w[0]/w[1])
                  intercept = -w[0]/w[2]
                  print("weights:", w, "Slope:", slope, "Intercept:", intercept)
                  x = np.array([-0.1, 1.1])
                  d = slope*x + intercept
                  plt.plot(x, d)

              plt.show()
```

b) Gráfico a seguir.

```
In [313]: plot_or(weights_history)
```

```
weights: [-0.8649  0.3092  0.3129] Slope: -0.988175135826 Intercept: 2.76414189837
weights: [-0.8649  0.3092  0.3129] Slope: -0.988175135826 Intercept: 2.76414189837
weights: [ 0.1351  0.3092  1.3129] Slope: -0.235509178155 Intercept: -0.102901972732
weights: [ 0.1351  0.3092  1.3129] Slope: -0.235509178155 Intercept: -0.102901972732
```

```
weights: [ 0.1351  0.3092  1.3129] Slope: -0.235509178155 Intercept: -0.102901972732
weights: [-0.8649  0.3092  1.3129] Slope: -0.235509178155 Intercept: 0.65877066037
weights: [-0.8649  0.3092  1.3129] Slope: -0.235509178155 Intercept: 0.65877066037
weights: [ 0.1351  1.3092  1.3129] Slope: -0.997181811258 Intercept: -0.102901972732
weights: [ 0.1351  1.3092  1.3129] Slope: -0.997181811258 Intercept: -0.102901972732
weights: [-0.8649  1.3092  1.3129] Slope: -0.997181811258 Intercept: 0.65877066037
weights: [-0.8649  1.3092  1.3129] Slope: -0.997181811258 Intercept: 0.65877066037
weights: [-0.8649  1.3092  1.3129] Slope: -0.997181811258 Intercept: 0.65877066037
weights: [-0.8649  1.3092  1.3129] Slope: -0.997181811258 Intercept: 0.65877066037
weights: [-0.8649  1.3092  1.3129] Slope: -0.997181811258 Intercept: 0.65877066037
```



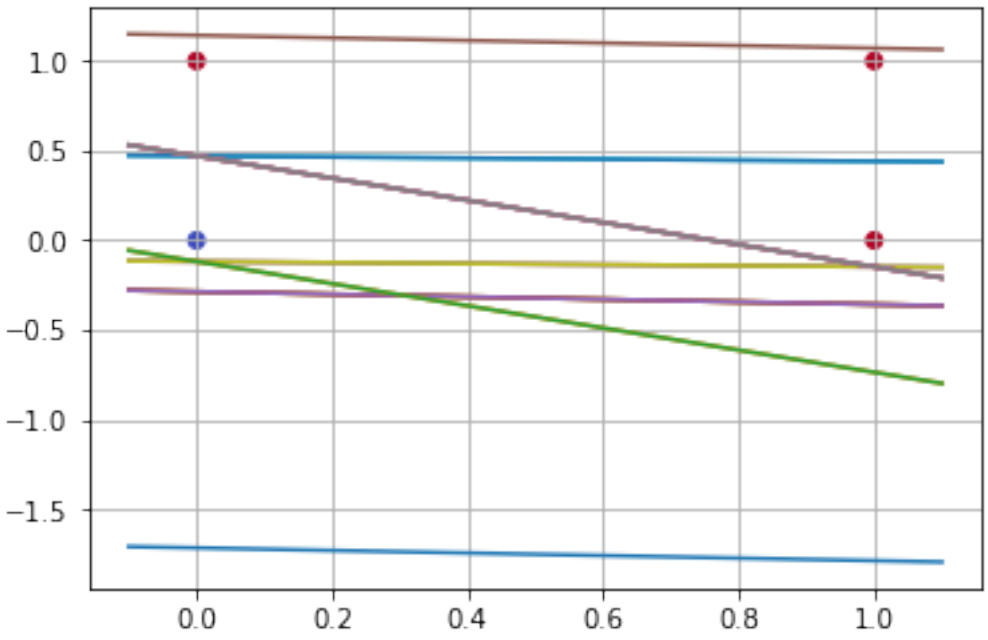
c) w1 = 0.05, w2 = 0.7, bias = 1.2

```
In [314]: perceptron = Perceptron([1.2, 0.05, 0.7])

In [315]: weights_history, errors_history = perceptron.fit(X, y)

In [316]: perceptron.predict(X)

Out[316]: array([0, 1, 1, 1])

In [317]: errors_history

Out[317]: [-1, 0, 0, 0, -1, 1, 0, 0, -1, 0, 1, 0, -1, 0, 0, 0, 0]
```

Convergiu em 13 passos.

```
In [318]: plot_or(weights_history)
```

```
weights: [ 1.2   0.05  0.7 ] Slope: -0.0714285714286 Intercept: -1.71428571429
weights: [ 0.2   0.05  0.7 ] Slope: -0.0714285714286 Intercept: -0.285714285714
weights: [ 0.2   0.05  0.7 ] Slope: -0.0714285714286 Intercept: -0.285714285714
weights: [ 0.2   0.05  0.7 ] Slope: -0.0714285714286 Intercept: -0.285714285714
weights: [ 0.2   0.05  0.7 ] Slope: -0.0714285714286 Intercept: -0.285714285714
weights: [-0.8   0.05  0.7 ] Slope: -0.0714285714286 Intercept: 1.14285714286
weights: [ 0.2   0.05  1.7 ] Slope: -0.0294117647059 Intercept: -0.117647058824
weights: [ 0.2   0.05  1.7 ] Slope: -0.0294117647059 Intercept: -0.117647058824
weights: [ 0.2   0.05  1.7 ] Slope: -0.0294117647059 Intercept: -0.117647058824
weights: [-0.8   0.05  1.7 ] Slope: -0.0294117647059 Intercept: 0.470588235294
weights: [-0.8   0.05  1.7 ] Slope: -0.0294117647059 Intercept: 0.470588235294
weights: [ 0.2   1.05  1.7 ] Slope: -0.617647058824 Intercept: -0.117647058824
weights: [ 0.2   1.05  1.7 ] Slope: -0.617647058824 Intercept: -0.117647058824
weights: [-0.8   1.05  1.7 ] Slope: -0.617647058824 Intercept: 0.470588235294
weights: [-0.8   1.05  1.7 ] Slope: -0.617647058824 Intercept: 0.470588235294
weights: [-0.8   1.05  1.7 ] Slope: -0.617647058824 Intercept: 0.470588235294
weights: [-0.8   1.05  1.7 ] Slope: -0.617647058824 Intercept: 0.470588235294
weights: [-0.8   1.05  1.7 ] Slope: -0.617647058824 Intercept: 0.470588235294
```



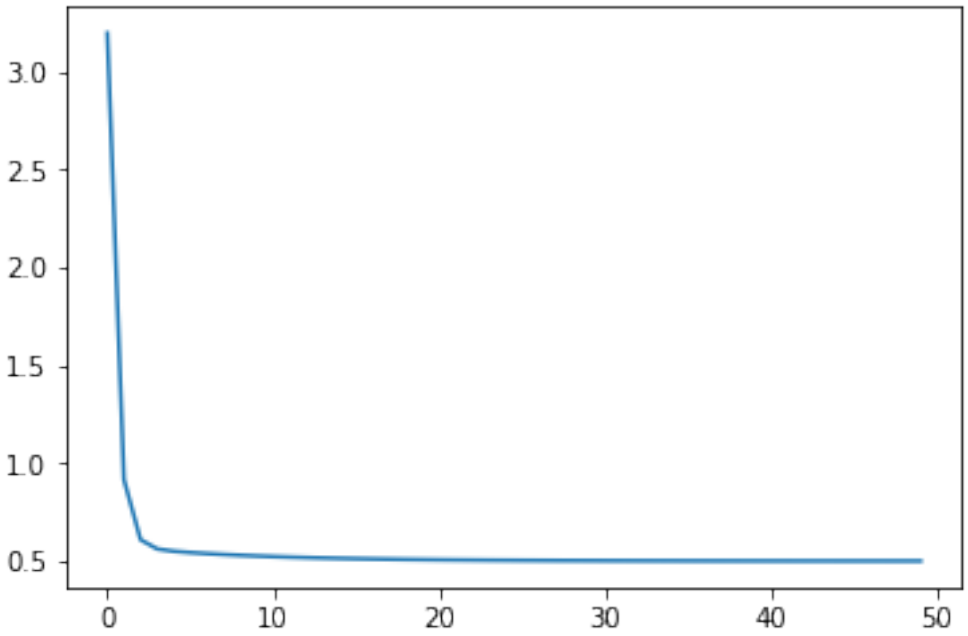
1.2.3 22. Faça a implementação do algoritmo de treinamento do neurônio Adaline. Realize o treinamento do neurônio para a solução do problema do operador lógico OU partindo da inicialização do pesos abaixo:

w1: 0.3092

w2: 0.3129

bias: -0.8649

```
In [421]: X = np.array([[0, 0],
                        [0, 1],
                        [1, 0],
                        [1, 1]])
          y = np.array([-1, 1, 1, 1])

In [457]: class Adaline:

              def __init__(self, initial_weights, learning_rate):
                  self._weights = np.array(initial_weights)
                  self._learning_rate = learning_rate

              def _prepend_bias(self, X):
                  return np.insert(np.atleast_2d(X), 0, values=1, axis=1)

              def _calculate_output(self, X):
                  return np.matmul(self._prepend_bias(X), self._weights)

              def predict(self, X):
                  return np.where(self._calculate_output(X) >= 0.0, 1, -1)

              def fit(self, X, y):
                  weights_history = [self._weights]
                  errors_history = []
                  for i in range (50):
                      output = self._calculate_output(X)
                      errors = (y - output)
                      delta = self._learning_rate * adaline._prepend_bias(X).T.dot(errors)
                      self._weights = self._weights + delta
                      weights_history.append(self._weights)
                      errors_history.append((errors**2).sum() / 2.0)

                  return weights_history, errors_history

In [458]: adaline = Adaline([-0.8649, 0.3092, 0.3129], 0.1)

In [459]: weights_history, errors_history = adaline.fit(X, y)

In [460]: adaline.predict(X)

Out[460]: array([-1,  1,  1,  1])

In [461]: plt.plot(np.arange(0, len(errors_history)), errors_history)

Out[461]: [<matplotlib.lines.Line2D at 0x7f4c9aaf5518>]
```

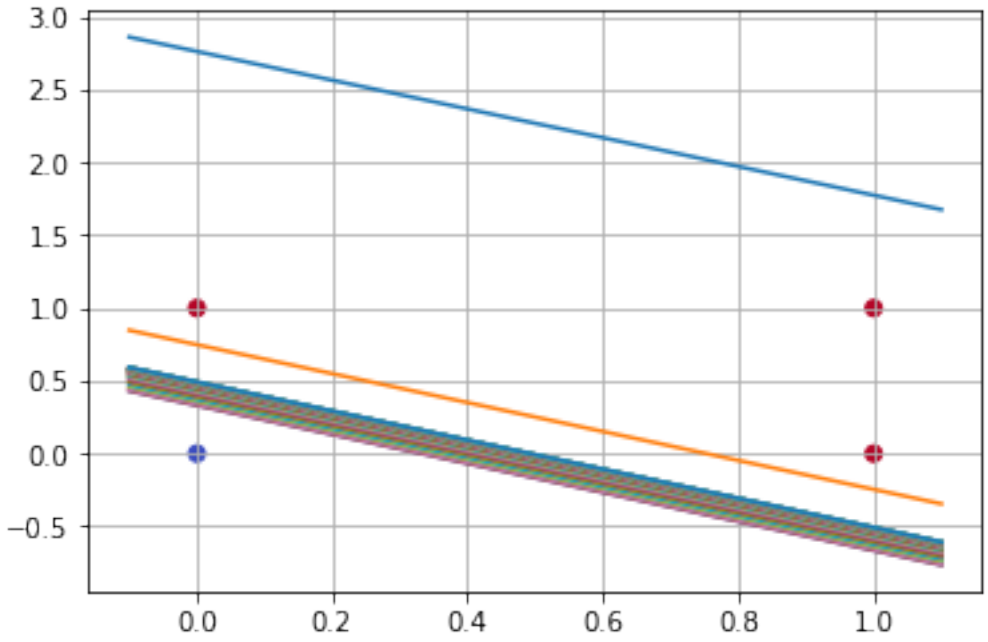


d) Por volta de 3 épocas o resultado já estava bem próximo de convergir à melhor solução.

e) Segue o gráfico com os hiperplanos de separação das 20 épocas de treinamento.

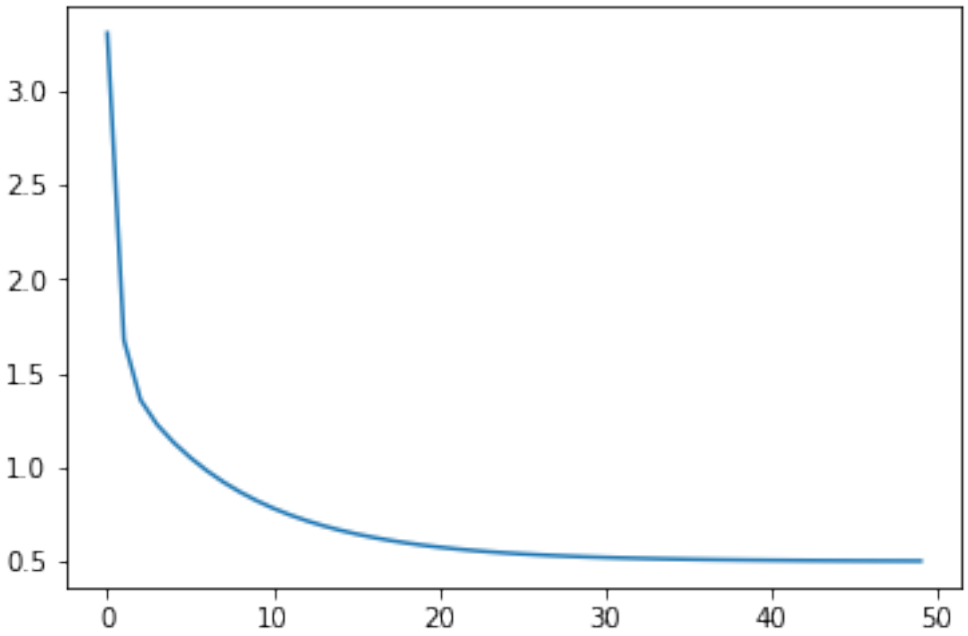
```
In [462]: plot_or(weights_history)

weights: [-0.8649  0.3092  0.3129] Slope: -0.988175135826 Intercept: 2.76414189837
weights: [-0.44336  0.58905  0.59238] Slope: -0.994378608326 Intercept: 0.748438502313
weights: [-0.302302  0.700674  0.703671] Slope: -0.995740907327 Intercept: 0.429607018052
weights: [-0.2622502  0.7506325  0.7533298] Slope: -0.996419496481 Intercept: 0.348121367295
weights: [-0.25814258  0.77762306  0.78005063] Slope: -0.996887932774 Intercept: 0.330930544855
weights: [-0.26642029  0.7957219   0.79790671] Slope: -0.997261819005 Intercept: 0.333899040233
weights: [-0.27857789  0.81007091  0.81203724] Slope: -0.997578520285 Intercept: 0.343060492131
weights: [-0.29156837  0.82256858  0.82433828] Slope: -0.997853188945 Intercept: 0.353699898698
weights: [-0.30432239  0.83393471  0.83552744] Slope: -0.998093744617 Intercept: 0.364227884535
weights: [-0.31648586  0.8444595   0.84589296] Slope: -0.998305393376 Intercept: 0.374144105805
weights: [-0.32796201  0.85427548  0.85556559] Slope: -0.998492096638 Intercept: 0.383327724751
weights: [-0.33874542  0.86345623  0.86461733] Slope: -0.998657094681 Intercept: 0.391786527912
weights: [-0.34886196  0.87205233  0.87309732] Slope: -0.998803123937 Intercept: 0.399568242392
weights: [-0.35834711  0.88010453  0.88104502] Slope: -0.998932528601 Intercept: 0.406729623847
weights: [-0.36723817  0.88764854  0.88849498] Slope: -0.999047331344 Intercept: 0.413326108524
weights: [-0.37557161  0.89471697  0.89547877] Slope: -0.999149285034 Intercept: 0.419408726158
weights: [-0.38338211  0.90134002  0.90202564] Slope: -0.999239913545 Intercept: 0.425023520652
weights: [-0.3907024   0.90754588  0.90816293] Slope: -0.999320545134 Intercept: 0.430211788942
weights: [-0.3975632   0.91336089  0.91391624] Slope: -0.999392340208 Intercept: 0.435010545345
weights: [-0.40399335  0.91880973  0.91930954] Slope: -0.999456314643 Intercept: 0.43945301023
```

```
weights: [-0.41001986  0.9239155   0.92436533] Slope: -0.999513359483 Intercept: 0.443569060284
weights: [-0.41566808  0.92869984  0.92910469] Slope: -0.999564257649 Intercept: 0.44738562569
weights: [-0.42096175  0.93318302  0.93354738] Slope: -0.999609698189 Intercept: 0.450927035349
weights: [-0.42592313  0.93738403  0.93771195] Slope: -0.99965028844 Intercept: 0.454215315782
weights: [-0.43057307  0.94132065  0.94161579] Slope: -0.999686564474 Intercept: 0.457270450064
weights: [-0.43493113  0.94500956  0.94527518] Slope: -0.999719000074 Intercept: 0.460110602642
weights: [-0.43901563  0.94846635  0.94870541] Slope: -0.999748014477 Intercept: 0.462752315084
weights: [-0.44284373  0.95170567  0.95192082] Slope: -0.999773979073 Intercept: 0.465210677046
weights: [-0.44643154  0.9547412   0.95493484] Slope: -0.999797223206 Intercept: 0.467499476042
weights: [-0.44979413  0.95758578  0.95776006] Slope: -0.999818039225 Intercept: 0.469631329037
weights: [-0.45294564  0.96025145  0.96040829] Slope: -0.999836686869 Intercept: 0.471617798413
weights: [-0.45589933  0.96274946  0.96289062] Slope: -0.9998533971 Intercept: 0.473469494428
weights: [-0.45866762  0.96509037  0.96521742] Slope: -0.999868375457 Intercept: 0.475196166005
weights: [-0.46126213  0.96728408  0.96739842] Slope: -0.999881804984 Intercept: 0.476806781364
weights: [-0.46369378  0.96933985  0.96944275] Slope: -0.999893848808 Intercept: 0.478309599829
weights: [-0.46597278  0.97126636  0.97135897] Slope: -0.999904652393 Intercept: 0.479712235912
weights: [-0.46810874  0.97307174  0.9731551 ] Slope: -0.999914345536 Intercept: 0.481021716646
weights: [-0.47011061  0.97476363  0.97483865] Slope: -0.999923044116 Intercept: 0.482244532987
weights: [-0.47198682  0.97634916  0.97641668] Slope: -0.999930851639 Intercept: 0.483386685987
weights: [-0.47374526  0.97783503  0.97789579] Slope: -0.999937860606 Intercept: 0.484453728371
weights: [-0.47539332  0.9792275   0.97928218] Slope: -0.99994415372 Intercept: 0.48545080203
weights: [-0.47693793  0.98053244  0.98058166] Slope: -0.999949804955 Intercept: 0.486382671908
weights: [-0.47838558  0.98175537  0.98179967] Slope: -0.999954880504 Intercept: 0.487253756673
weights: [-0.47974236  0.98290145  0.98294132] Slope: -0.999959439618 Intercept: 0.488068156538
weights: [-0.48101397  0.9839755   0.98401138] Slope: -0.999963535353 Intercept: 0.488829678526
weights: [-0.48220575  0.98498205  0.98501435] Slope: -0.999967215234 Intercept: 0.489541859455
weights: [-0.48332273  0.98592536  0.98595442] Slope: -0.999970521844 Intercept: 0.490207986883
weights: [-0.4843696   0.98680939  0.98683555] Slope: -0.999973493348 Intercept: 0.49083111822
weights: [-0.48535075  0.98763788  0.98766142] Slope: -0.999976163961 Intercept: 0.491414098186
weights: [-0.48627031  0.98841431  0.9884355 ] Slope: -0.999978564365 Intercept: 0.491959574787
weights: [-0.48713215  0.98914196  0.98916103] Slope: -0.999980722079 Intercept: 0.492470013955
```



f) w1 = 0.05, w2 = 0.7, bias = 1.2

```
In [463]: adaline = Adaline([1.2, 0.05, 0.7], 0.1)

In [464]: weights_history, errors_history = adaline.fit(X, y)

In [465]: adaline.predict(X)

Out[465]: array([-1,  1,  1,  1])

In [466]: plt.plot(np.arange(0, len(errors_history)), errors_history)

Out[466]: [<matplotlib.lines.Line2D at 0x7f4c9b183828>]
```

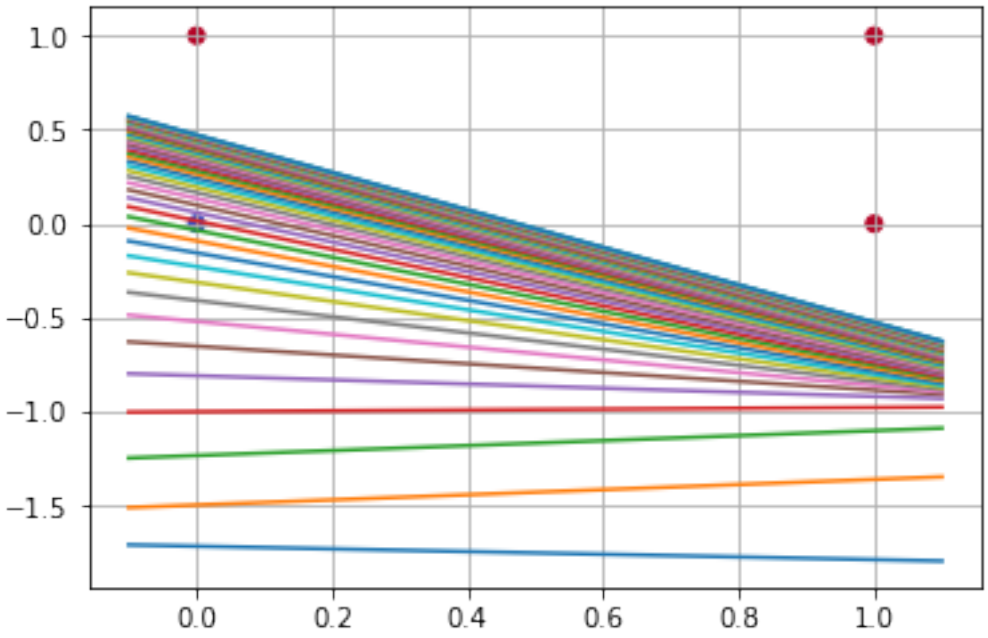


Nesse caso, já demorou mais para convergir, levando ao menos 20 épocas.

```
In [467]: plot_or(weights_history)
```

```
weights: [ 1.2    0.05   0.7 ] Slope: -0.0714285714286 Intercept: -1.71428571429
weights: [ 0.77   -0.07    0.515] Slope: 0.135922330097 Intercept: -1.49514563107
weights: [ 0.573   -0.0615   0.465 ] Slope: 0.132258064516 Intercept: -1.23225806452
weights: [ 0.4631   -0.0103    0.46355] Slope: 0.0222198252616 Intercept: -0.999029230935
weights: [ 0.38721    0.052785   0.47925 ] Slope: -0.11014084507 Intercept: -0.807949921753
weights: [ 0.325919    0.116861    0.5006795] Slope: -0.233404802873 Intercept: -0.650953354391
weights: [ 0.2720433    0.17823705   0.5236737 ] Slope: -0.340358986903 Intercept: -0.519490094691
weights: [ 0.22284383   0.23581361   0.54670659] Slope: -0.431334855216 Intercept: -0.407611380653
weights: [ 0.17720226   0.28941146   0.56921515] Slope: -0.50843949429 Intercept: -0.311309804933
weights: [ 0.13459603   0.3391672    0.59099052] Slope: -0.573896181635 Intercept: -0.22774651537
weights: [ 0.09472607   0.3853155    0.61195649] Slope: -0.629645261408 Intercept: -0.154792171604
weights: [ 0.05738125   0.42811154   0.63208843] Slope: -0.677296911788 Intercept: -0.090780408259
weights: [ 0.02238875   0.46780414   0.65138334] Slope: -0.718170257325 Intercept: -0.034371087796
weights: [-0.01040424   0.50462723   0.66984851] Slope: -0.753345305203 Intercept: 0.0155322334328
weights: [-0.04113769   0.53879778   0.68749693] Slope: -0.783709360313 Intercept: 0.0598369110577
weights: [-0.06994156   0.57051607   0.70434531] Slope: -0.809994849306 Intercept: 0.0993000979098
weights: [-0.09693721   0.59996664   0.72041295] Slope: -0.832809345702 Intercept: 0.134557839008
weights: [-0.12223824   0.62731946   0.73572114] Slope: -0.852659281417 Intercept: 0.166147520601
weights: [-0.14595106   0.6527311    0.75029261] Slope: -0.869968714498 Intercept: 0.194525524998
weights: [-0.16817538   0.67634583   0.76415119] Slope: -0.885094255641 Intercept: 0.220081291085
weights: [-0.18900463   0.69829662   0.77732145] Slope: -0.898337006229 Intercept: 0.243148615295
weights: [-0.20852639   0.71870608   0.78982842] Slope: -0.909952160619 Intercept: 0.264014801376
weights: [-0.22682274   0.7376873    0.80169741] Slope: -0.920156772456 Intercept: 0.282928115254
```

```
weights: [-0.24397058  0.75534465  0.81295374] Slope: -0.929136069342 Intercept: 0.300103893536
weights: [-0.26004203  0.77177446  0.82362265] Slope: -0.937048613047 Intercept: 0.315729575132
weights: [-0.27510464  0.78706571  0.83372908] Slope: -0.944030536495 Intercept: 0.329968866326
weights: [-0.28922174  0.80130059  0.84329762] Slope: -0.950199038485 Intercept: 0.342965204628
weights: [-0.30245269  0.81455506  0.85235238] Slope: -0.955655278583 Intercept: 0.35484465222
weights: [-0.3148531   0.82689934  0.86091694] Slope: -0.960486784971 Intercept: 0.365718323144
weights: [-0.32647512  0.8383984   0.86901424] Slope: -0.964769464956 Intercept: 0.37568442754
weights: [-0.3373676   0.84911232  0.87666657] Slope: -0.96856928993 Intercept: 0.384829999997
weights: [-0.34757634  0.85909672  0.88389555] Slope: -0.971943712454 Intercept: 0.393232366168
weights: [-0.35714425  0.86840309  0.89072203] Slope: -0.974942862053 Intercept: 0.400960391695
weights: [-0.36611158  0.87707912  0.89716617] Slope: -0.97761055748 Intercept: 0.408075549324
weights: [-0.374516    0.88516899  0.90324734] Slope: -0.97998516624 Intercept: 0.414632833625
weights: [-0.38239287  0.89271366  0.90898417] Slope: -0.982100336521 Intercept: 0.420681547525
weights: [-0.38977529  0.89975109  0.91439454] Slope: -0.983985622196 Intercept: 0.426265980632
weights: [-0.3966943   0.90631647  0.91949558] Slope: -0.985667017928 Intercept: 0.431425995914
weights: [-0.40317899  0.91244248  0.92430368] Slope: -0.987167418446 Intercept: 0.436197538525
weights: [-0.40925663  0.91815941  0.92883449] Slope: -0.988507013697 Intercept: 0.440613078274
weights: [-0.41495276  0.92349541  0.93310298] Slope: -0.989703629587 Intercept: 0.444701995403
weights: [-0.42029133  0.92847658  0.93712339] Slope: -0.990773022452 Intercept: 0.448490917738
weights: [-0.42529479  0.93312719  0.94090932] Slope: -0.991729134073 Intercept: 0.452004016086
weights: [-0.42998418  0.93746978  0.9444737 ] Slope: -0.992584312947 Intercept: 0.455263263621
weights: [-0.4343792   0.94152529  0.94782882] Slope: -0.993349506652 Intercept: 0.458288664178
weights: [-0.43849834  0.94531319  0.95098637] Slope: -0.994034429382 Intercept: 0.461098453607
weights: [-0.44235892  0.94885158  0.95395744] Slope: -0.994647708102 Intercept: 0.463709277754
weights: [-0.44597715  0.95215731  0.95675258] Slope: -0.99519701026 Intercept: 0.466136350085
weights: [-0.44936827  0.95524602  0.95938176] Slope: -0.995689155563 Intercept: 0.468393591577
weights: [-0.45254652  0.95813229  0.96185446] Slope: -0.996130213949 Intercept: 0.470493755093
weights: [-0.45552526  0.96082969  0.96417964] Slope: -0.996525591571 Intercept: 0.472448536183
```



1.2.4 23. Explique a figura abaixo: (PDF)

O algoritmo de treinamento do Perceptron descartar informação ao utilizar a função de ativação **step**. Dessa forma, assim que encontra uma solução que classifique corretamente o dataset de treinamento, ele para os ajustes. Isso faz com que diferentes retas sejam geradas dependendo dos valores iniciais dos pesos. O problema disso é que ele tende a chegar em soluções que não generalizam tão bem. Se o testarmos com um dataset não utilizado durante os testes, ele terá mais chances de cometer erros.

Por outro lado, o algoritmo de treinamento do Adaline utiliza a saída da rede (antes de quantizar a saída) para obter o erro da rede. Além disso, utiliza um algoritmo de otimização baseado em gradiente descendente. Dessa forma, com número suficiente de épocas e taxa de aprendizagem correta, ele sempre encontrará a mesma curva, que melhor separa os dois conjuntos através da minimização do erro quadrático médio. Isso faz com que se alcance uma generalização melhor, reduzindo os erros em um dataset de testes.