UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Instituto de Física
Bacharelado em Engenharia Física
Tópicos especiais de engenharia física

Deomar Santos da Silva Junior / 00260682

Implementação de rede neural para o treinamento de uma porta *XOR*

Introdução

Redes neurais são modelos computacionais baseados no funcionamento dos neurônios cerebrais [1]. A rede é basicamente composta por nós (neurônios) - figura 1, que são funções matemáticas não lineares (funções *h*) com pesos que aceitam entradas (A e B) e produzem saídas no neurônio de saída (função *g*). Comparando-se as saídas desejadas com as saídas produzidas por pesos iniciais aleatórios, pode-se modificar os pesos de forma a treinar a rede, ou seja, escolher os pesos que mais aproximam as saídas desejadas.

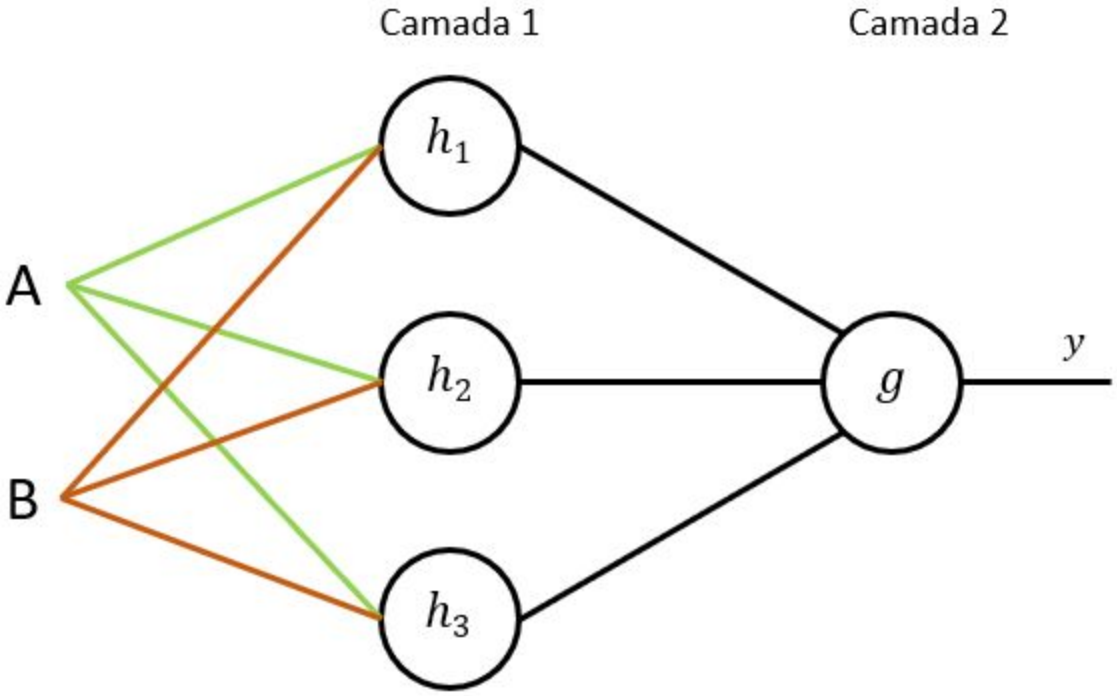

Figura 1. Rede neural de duas camadas com 4 neurônios.

Para encontrar os pesos que aproximam as saídas, minimiza-se uma função erro que mensura quanto as saídas calculadas estão "longe" das saídas desejadas. Então, utiliza-se a técnica do gradiente descendente em relação a cada um dos pesos para ajustar os pesos na direção de minimização. O que diferencia uma rede neural de uma multiplicação simples de matrizes lineares é o teorema da aproximação universal [2] que, em palavras intuitivas e simplificadas, afirma que um conjunto de neurônios com funções *não lineares* pode aproximar qualquer função contínua.

Feedforward

O processo de cálculo das saídas é chamado de *feedforward* [3], que pode ser entendido como um processo de alimentação 'para frente', ou seja, um processo de cálculo das camadas subsequentes em função das camadas anteriores.

Matematicamente, tem-se:

Dado que a função não linear é uma função sigma (σ):

$$\sigma = 1/(1 + e^{-s})$$

Em que *s* são as entradas do neurônio.

Como se deseja introduzir pesos que possam ser ajustados, as entradas desse neurônio são multiplicadas pelos pesos '*w*' de forma que:

$$s = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$$

O processo é o mesmo para os outros neurônios. A única diferença é que cada neurônio tem os seus próprios pesos 'w'.

Para a camada 1, indicada pelo índice 1, as entradas de todos os neurônios podem ser escritas pelo produto escalar:

$$S_1 = W . X$$

Onde W é a matriz de pesos dos neurônios da camada 1 e X é a matriz de entrada.

Logo, para os neurônios da camada 1, tem-se:

$$H = \sigma(S_1)$$

Sendo H a matriz saída da camada 1.

Semelhantemente para os neurônios da camada 2:

$$G = \sigma(S_2) = y$$

Sendo que $S_2 = U \,.\, H$, a matriz U é a matriz dos pesos da segunda camada, a matriz H é a saída dos neurônios da camada 1 e ***y*** é a saída calculada.

Função erro

Como se quer tornar as saídas calculadas o mais próximas das saídas desejadas, matematicamente introduz-se a função erro que calcula "quão longe" estão as duas. Uma forma fácil é medir o erro, ou seja, a soma da diferença quadrática dada por:

$$L = \sum_{i=1}^{n} (y - \hat{y})^2$$

Em que agora y é a saída desejada $\hat{y}$ é a saída calculada. Com a função erro, basta minimizá-la, ou seja, calcular o seu gradiente, em relação aos parâmetros que se quer ajustar. Entretanto, como se quer ajustar em relação aos pesos das camadas anteriores, o processo de minimização engloba a derivada parcial em que se utiliza a regra da cadeia para calcular as diferenças finitas das funções dos neurônios seguintes em relação aos neurônios anteriores.

Backpropagation

Após calcular as derivadas parciais para a minimização, ajusta-se os pesos através do gradiente. A esse processo inteiro de ajuste se dá o nome de *backpropagation*.

Então, minimizando a função de perda em relação aos pesos ***u*** da camada 2:

$$\partial L/\partial U = (\partial L/\partial \hat{y})\partial \hat{y}/\partial S_2 \,.\, \partial S_2/\partial U$$

De forma que:

$$\partial L/\partial \hat{y} = 2(y - \hat{y})$$

$$\partial \hat{y}/\partial S_2 = [\sigma(\hat{y})][1 - \sigma(\hat{y})]$$

$$\partial S_2/\partial U = H$$

Substituindo todas as derivadas:

$$\partial L/\partial U = 2(y - \widehat{y})[\sigma(\widehat{y})][1 - \sigma(\widehat{y})] \; . \; H$$

Semelhantemente para os pesos ***w***:

$$\partial L/\partial W = [(\partial L/\partial \widehat{y})\partial \widehat{y}/\partial S_2] \; . \; [\partial S_2/\partial H(\partial H/\partial S_1)] \; . \; \partial S_1/\partial W$$

Além dos termos já calculados acima, tem-se:

$$\partial S_2/\partial H \; = U$$

$$\partial H/\partial S1 \; = [\sigma(S_1)][1 - \sigma(S_1)]$$

$$\partial S1/\partial W \; = X$$

Substituindo:

$$\partial L/\partial W = \{2(y - \widehat{y})[\sigma(\widehat{y})][1 - \sigma(\widehat{y})]\} \; . \; \{U[\sigma(S_1)][1 - \sigma(S_1)]\} \; . \; X$$

Calculados os gradientes em relação aos pesos, atualiza-se os pesos por iterações sucessivas no tempo, de ***t*** para ***t + 1,*** através do método gradiente de forma que:

$$U_{t+1} = U_t + \alpha \; \partial L/\partial U$$

$$W_{t+1} = W_t + \alpha \; \partial L/\partial W$$

Onde α é a taxa de otimização do gradiente que tem o objetivo de aproximar a função a passos pequenos de forma a não dar "saltos largos" entre o mínimo da função.

Implementação de um neurônio

Primeiramente, para simplificação da metodologia, implementou-se um neurônio em *python* com programação orientada a objetos (programa [*Trabalho 7 - 1 neuronio - Deomar.py]*) de apenas uma saída que aceita duas entradas (figura 2).

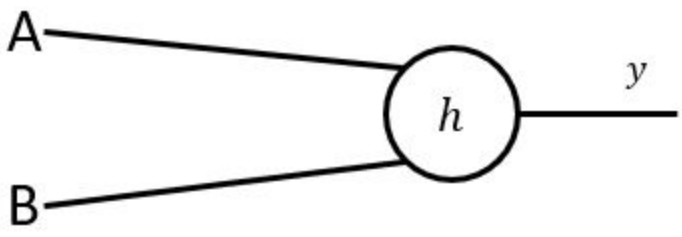

Figura 2. Rede neural de um neurônio.

De forma que:

$$h = y = \sigma(w1A \ + \ w2B \ + \ b)$$

$$X = (A, B)$$

$$W = (w1, w2)$$

E a mesma função erro:

$$L = \sum_{i=1}^{n} (y - \widehat{y})^2$$

Seguindo a mesma lógica das derivações parciais acima, tem-se para os pesos ***w***:

$$\partial L/\partial W \ = 2(y - \widehat{y})[\sigma(X)][1 - \sigma(X)] \ . \ X$$

Semelhantemente para b:

$$\partial L/\partial b \ = 2(y - \widehat{y})[\sigma(b)][1 - \sigma(b)]$$

As equações de atualização dos pesos ficam:

$$W_{t+1} = W_t + \alpha \ \partial L/\partial U$$

$$b_{t+1} = b_t \ + \alpha \ \partial L/\partial b$$

Essa construção simplifica bastante as equações do modelo uma vez que não é necessário realizar as derivadas parciais dos neurônios de uma camada em relação às outras camadas.

Para esse teste simplificado, utilizou-se a porta AND e OR para a entrada X:

X = (0,1)

e:

Y = 1 para a porta OR

Y = 0 para a porta AND

Implementando-se as equações acima para uma taxa (α) de 0.01 e 5000 iterações (*epochs*) na porta OU, tem-se, para a função de erro:

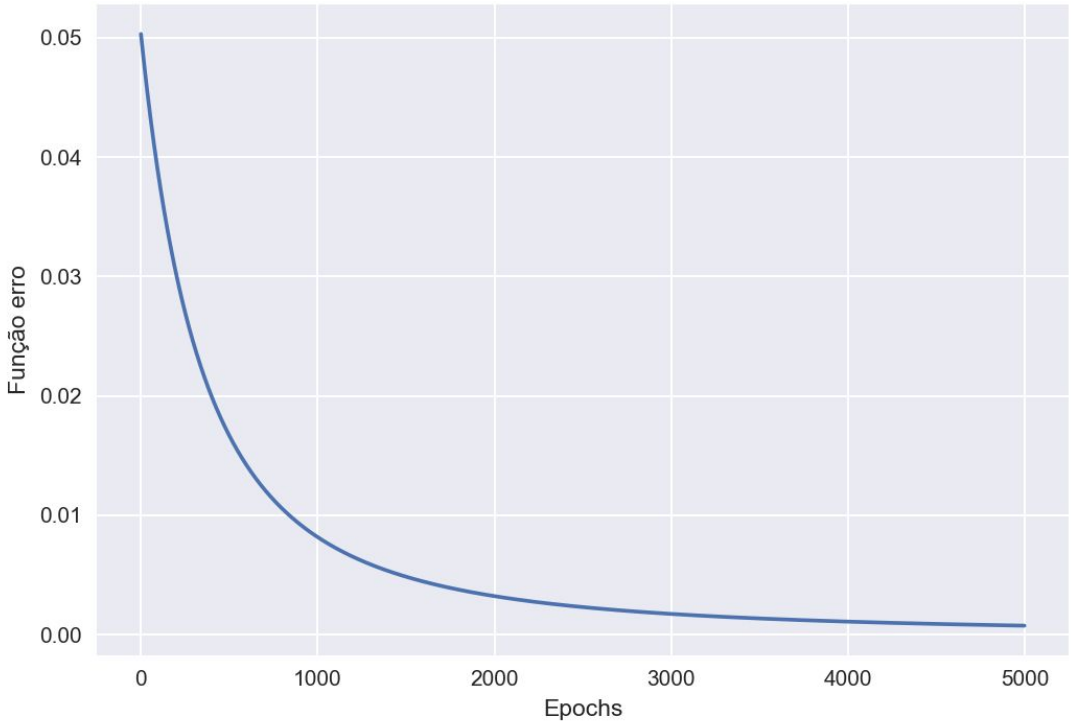

Figura 3. Função erro na rede de um neurônio.

A função de erro para a porta AND tem comportamento semelhante.

E a saída da rede neural para a porta OR:

Y = 0.96502003

E para a porta AND:

Y = 0.01584266

Implementação da porta *XOR* com uma rede de 4 neurônios

Finalmente, para o treinamento de uma porta XOR (tabela 1) implementa-se uma rede neural (programa [*Trabalho 7 - Rede 4 neuronios - Deomar.py]*) com 4 neurônios (figura 1). As equações foram deduzidas na explicação teórica do *feedforward* e *backpropagation*.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Tabela 1. Porta XOR.

Treina-se o modelo com as entradas A e B e a saída Y com uma taxa de (α) de 0.1 e 40000 iterações. A função erro calculada em relação ao número de iterações é mostrada na figura 4.

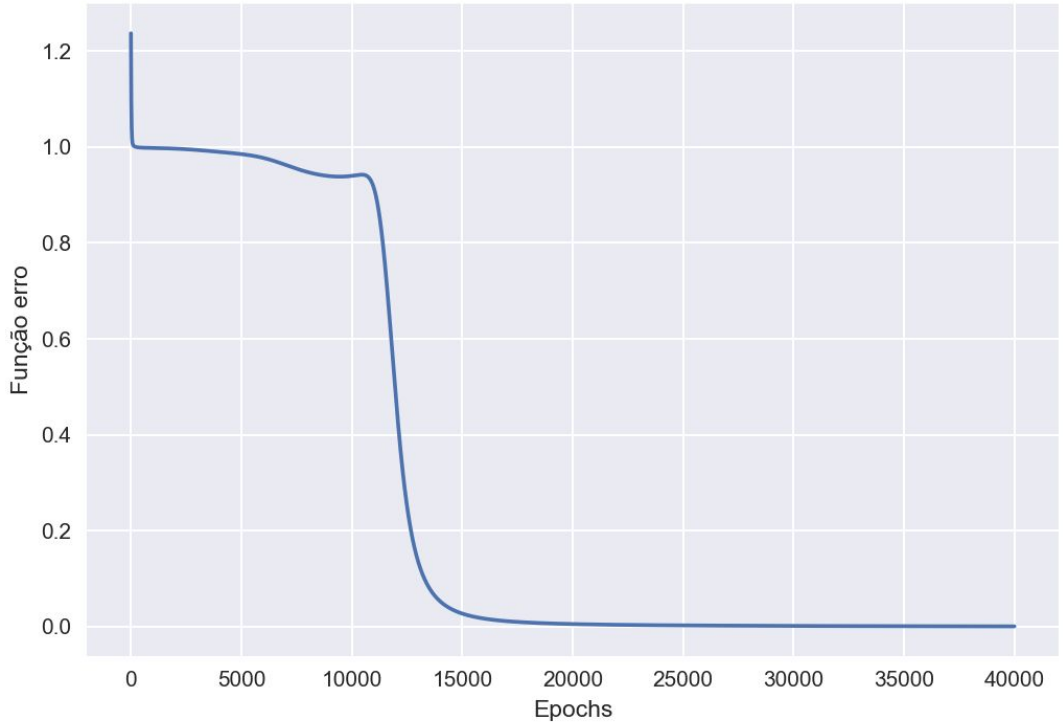

Figura 4. Função erro na rede de quatro neurônios.

Com as saídas Y:

$$Y = \begin{matrix}[0.00947103] \\ [0.98197638]\end{matrix}$$

[0.98245666]

[0.01488674]

Conclusão

Foi possível implementar, utilizando um exemplo de entrada, portas mais simples como as portas AND e OR com apenas um neurônio que envolve equações simplificadas da teoria de redes neurais. E, então, foi possível demonstrar a aplicação de uma rede neural mais robusta com 4 neurônios para a implementação da porta XOR.

Referências:

[1] **https://en.wikipedia.org/wiki/Artificial_neural_network** **<Disponível em 03/08/2020>**

[2] **https://en.wikipedia.org/wiki/Universal_approximation_theorem**

<Disponível em 03/08/2020>

[3][**https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6**](https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6) **<Disponível em 03/08/2020>**