Machine Learning
2022

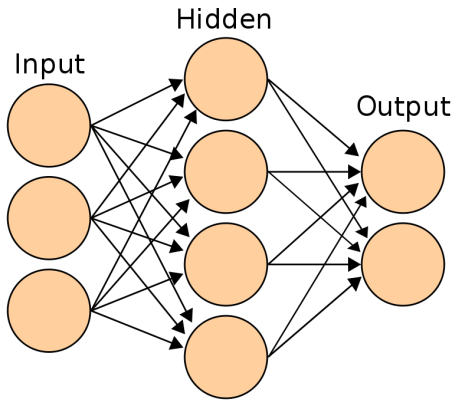
PROFESSOR: Arthur Rocha
prof.arthur.rocha@gmail.com

O que são Redes Neurais Artificiais?

São modelos de machine learning inspirados no sistema nervoso biológico (cérebro, neurônios, sinapses).

Por exemplo, uma rede neural para o reconhecimento de escrita manual é definida por um **conjunto de neurônios de entrada** que podem ser ativados pelos pixels de uma imagem. Os dados adquiridos por essa ativação dos neurônios são então repassados, ponderados e transformados por uma função (chamada de **função de ativação**), a outros neurônios. Este processo é repetido até que, finalmente, um **neurônio de saída é ativado**. Isso determina que caractere foi lido a partir dessa imagem de entrada.

Redes neurais têm sido usadas para resolver uma grande variedade de tarefas que são difíceis de resolver utilizando programação baseada em regras comuns, incluindo **visão computacional** e **reconhecimento de voz**.



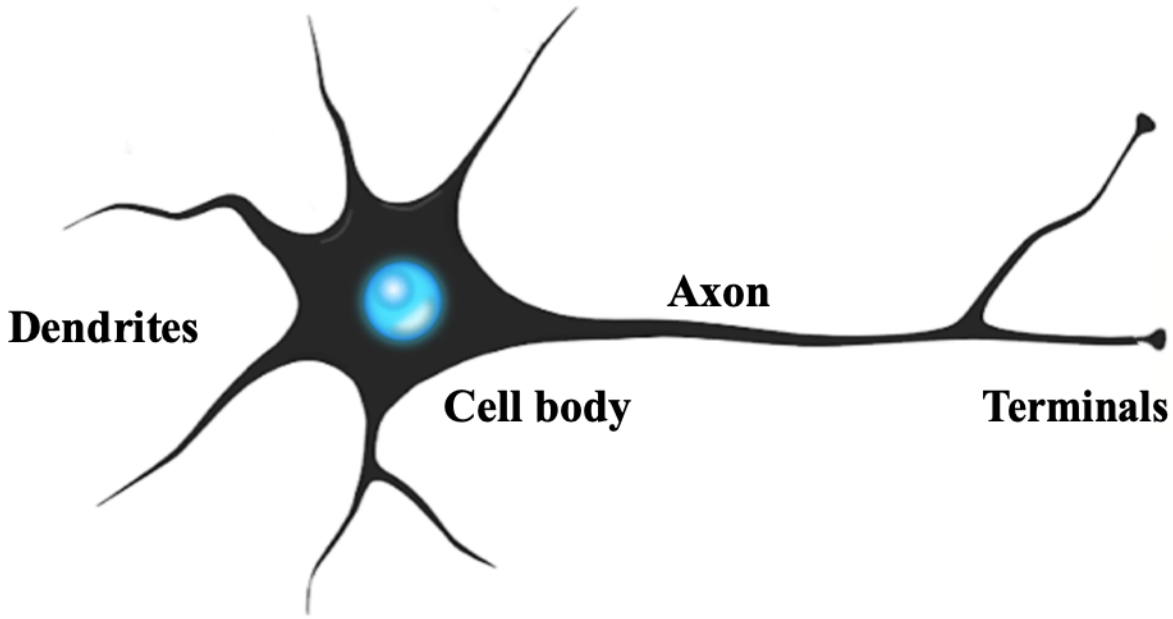
O que seria o “neurônio artificial”?

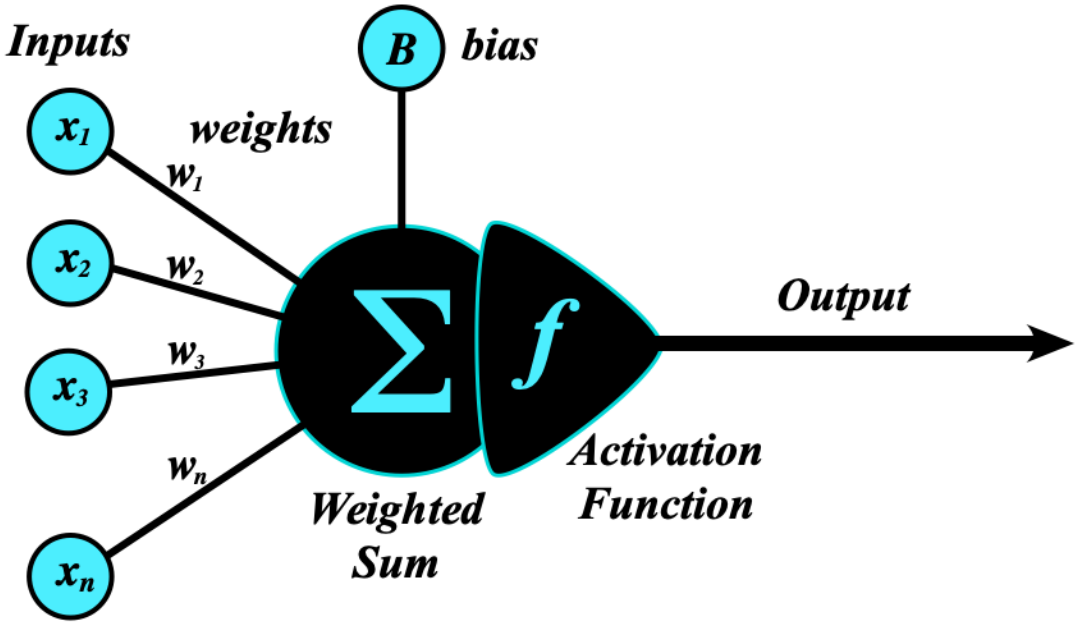
O neurônio é a unidade mínima que compõe toda a rede neural. Em outras palavras, a rede neural não passa de um aglomerado de neurônios conectados por **pesos sinápticos** e **funções de ativação**.

Mas o que isso representa computacionalmente? Na figura abaixo temos um desenho de um modelo de neurônio biológico (com suas estruturas básicas) e um paralelo abaixo com um modelo computacional desse neurônio. Importante! Tenha em mente que o modelo computacional do neurônio é uma versão extremamente simplificada de um neurônio biológico real e na prática são coisas completamente diferentes.

Quando dizemos que o neurônio computacional foi inspirado no biológico, queremos dizer que ele recebe valores de entrada (vetor de valores x_1, x_2, x_3 ... x_n), processa eles multiplicando cada um desses valores pelos pesos sinápticos correspondentes (w_1, w_2, w_3 ... w_n). Ao final desse processo, somam-se todos os valores, ou seja: $\sum_{i=1}^{n} w_i x_i$ e aplicamos esse resultado a uma função de ativação que costuma ser uma função não linear simples (veremos mais a frente).

Note que a fórmula do somatório que escrevemos no parágrafo anterior, pode ser reescrita de uma maneira até mais simples. Se considerarmos que as nossas entradas formam um vetor **X** e os pesos formam um vetor **W**, então o somatório é a mesma coisa que o simples produto entre esses dois vetores: **WX**. E ainda mais, quando falamos de redes neurais que se organizam em camadas que combinam vários desses neurônios, então para cada uma dessas camadas temos não apenas um vetor de pesos sinápticos, mas uma matriz bidimensional. Talvez agora tenha ficado mais claro porque estudamos álgebra linear com um foco em produto matricial.


Dendrites
Axon
Cell body
Terminals

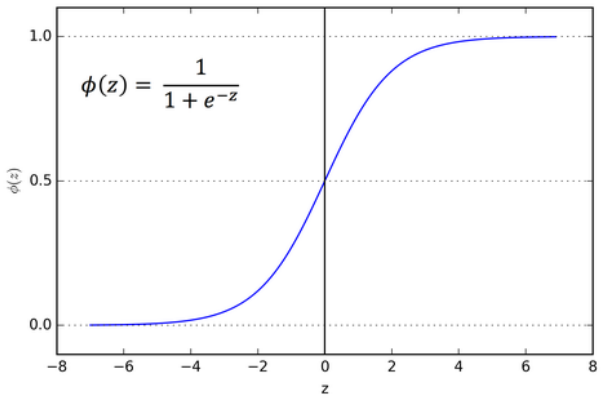

Inputs
B
bias
x1
weights
w1
x2
w2
w3
x3
wn
xn
Σ
f
Output
Activation
Function
Weighted
Sum

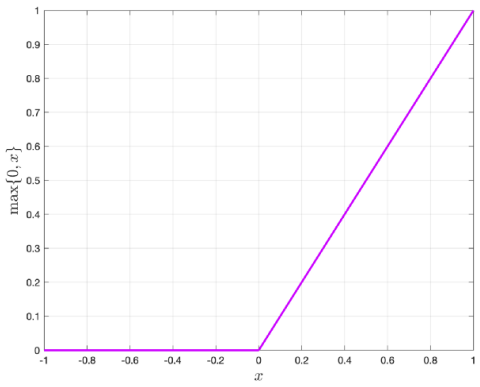
Funções de Ativação

Funções de ativação, de uma maneira bem simplificada, são funções não-lineares bem simples aplicadas a cada um dos neurônios que permitem que a rede neural possua uma melhor capacidade de aproximar funções mais complexas como um todo. E isso leva a uma melhor capacidade de inferência e predição do nosso modelo de machine learning.

Existem diversas funções de ativação desde a mais simples de todas (um simples limiar - threshold) passando por algumas funções como a tangente hiperbólica.

Duas funções muito utilizadas hoje em dia e que vamos comentar aqui são a função **sigmóide** (lado esquerdo da imagem abaixo) e a função **ReLU** (Rectified Linear Unit - lado direito da imagem abaixo). A função sigmóide (também chamada função logística) pode ser enxergada como uma aproximação contínua e diferenciável (cálculo diferencial) da função de limiarização, ou seja, para valores de entrada maiores que zero, a função retorna valores que vão se aproximando de um; e para valores de entrada menores que zero, a função retorna valores que se aproximam de zero. A função tangente hiperbólica é quase idêntica, com a diferença de retornar valores entre [1, -1] em vez de [1, 0]. Por muito tempo, essas funções dominaram, mas mais recentemente a função ReLU tem sido muito mais utilizada. A função ReLU é mais simples de ser calculada (é apenas o resultado de $y = max(0, x)$). Não está muito claro como, mas essa função funciona extremamente bem na prática e é muito simples computacionalmente de computar. Também se utilizam variações dessa função como a Leaky ReLU, entre outras.





Como treinar Redes Neurais?

Redes Neurais, assim como todos os outros modelos de machine learning não seriam muito úteis (e nem poderiam ser chamados de machine learning) se não houvesse uma maneira de fazer com que esses modelos sejam capazes de aprender, ou seja, extrair (a partir dos dados) funções e maneiras de fazer inferências e extrapolar o conhecimento para dados futuros desconhecidos.

Não é diferente com as redes neurais, mas antes de falarmos sobre o treinamento delas, é importante mencionar que no início, por um grande momento, as redes neurais foram desacreditadas, pois as que existiam na época eram muito simples e não conseguiam resolver problemas muito triviais. E, construções mais complexas de redes neurais (que seriam efetivamente capazes de resolver problemas mais complexos) não eram possíveis de serem treinadas antes da invenção do algoritmo de **Backpropagation**.

Backpropagation

Esse algoritmo foi uma revolução e é baseado na chamada **regra da cadeia** (teorema muito importante de cálculo diferencial).

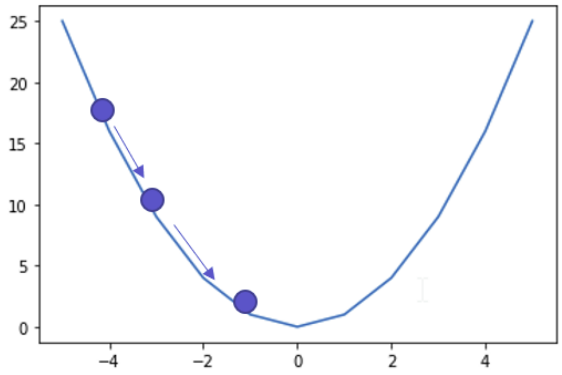
Não vamos entrar na parte matemática desse processo, pois precisaríamos de uma introdução mesmo que superficial ao cálculo diferencial, derivadas e mais alguns conceitos. Porém, de uma maneira bem resumida, o algoritmo de backpropagation se caracteriza por (como o próprio nome já sugere) propagar o erro de classificação/regressão a partir da última camada de neurônios (camada de saída) e progressivamente ir compartilhando a proporção de contribuição desse erro para cada uma das camadas mais internas até a primeira camada.

Então, quando estamos treinando a nossa rede neural, temos duas fases. Na primeira fase, também chamada **feedforward**, a informação trafega da entrada para a saída. Depois, o erro de predição é calculado como sendo a diferença entre o que foi computado nessa fase feedforward e o que era esperado. Esse erro é utilizado para a segunda fase, onde o algoritmo de **backpropagation** é aplicado e, como já descrito, segue o caminho da informação inverso ao da fase de feedforward.

Gradiente Descendente

O algoritmo de backpropagation define de que maneira a informação do erro de predição é propagada de maneira retrospectiva e proporcional para cada um dos neurônios que compõem a rede neural. E o algoritmo de **gradiente descendente** é quem diz como que os valores dos pesos sinápticos (parâmetros de aprendizado da rede neural) são atualizados a cada uma das iterações do treinamento da rede neural.

O gradiente descendente é um algoritmo que se comporta de maneira similar a um "objeto" que desce uma montanha por efeito da gravidade (figura abaixo).



Mais uma vez, para compreendermos completamente como funciona esse algoritmo (equação abaixo), temos que ter uma base mínima de cálculo diferencial, mas podemos ter uma intuição sem precisar de tanta base matemática.

α = step; θ_n = parâmetros do modelo no instante n;
$F(\theta)$ = Função de custo; ∇ = Gradiente;

$$\theta_{n+1} = \theta_n - \alpha \nabla F(\theta)$$

De maneira mais resumida, na equação acima, estamos tentando atualizar todos os parâmetros da nossa rede neural (θ - todas as sinapses dos neurônios) a partir do tempo **n** para o tempo **n + 1**. E para isso, apenas subtraímos o termo da direita da equação. O fato

de estarmos subtraindo em vez de somar é porque o gradiente é positivo no sentido da subida e como estamos querendo descer, por isso subtraímos. O termo α serve para deixar o tamanho do passo menor e evitar instabilidades no nosso algoritmo (uma analogia seria para evitar que o nosso modelo desse um passo maior que a “perna”). Ao final, estamos computando o gradiente (derivada, taxa de variação, inclinação...) da função que queremos efetivamente minimizar, que é a chamada função de custo e basicamente simboliza o erro geral da nossa rede neural para todos os dados do nosso conjunto de treinamento.

No pytorch, tensorflow, ou qualquer outra lib de deep learning, temos todas essas funcionalidades de backpropagation, gradiente descendente (e muitos outros métodos de otimização) já implementados, bem como todas as computações de derivadas e framework de cálculo diferencial já pronto e completamente transparente para o usuário da lib. Então, do mesmo jeito que o numpy expande o python puro para trabalhar com álgebra linear e outras funções de computação científica/matemática, essas libs de deep learning expandem essas funcionalidades e acrescenta todo esse framework para lidar com treinamento de redes neurais.