Redes Neurais, Perceptron Multicamadas e o Algoritmo Backpropagation

Tiago M. Leite May 10

Você já se perguntou como funcionam os sistemas de reconhecimento de imagem? Como um aplicativo do seu celular faz para detectar rostos, ou um teclado inteligente sugere a próxima palavra? As chamadas Redes Neurais tem sido amplamente usadas para tarefas como essas, mas mostraram-se úteis também em outras áreas, como aproximação de funções, previsão de séries temporais e processamento de linguagem natural.

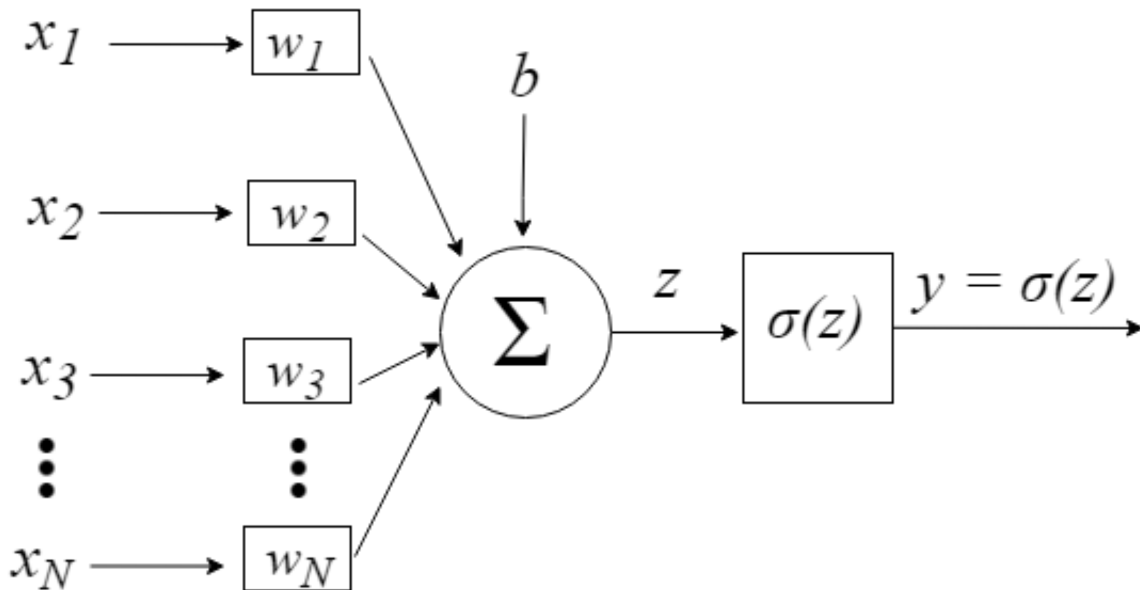
Neste artigo, explico como funciona um tipo básico de Rede Neural, o **Perceptron Multicamadas**, e um fascinante algoritmo responsável pelo aprendizado da rede, o **backpropagation**. Tal modelo de rede serviu de base para os modelos mais complexos hoje existentes, como as Redes Convolucionais, que são o estado da arte para classificação de imagens.

Se você estudou **Cálculo** na universidade e ainda não se esqueceu, não terá dificuldade em entender a parte do texto que está escrita em *Matematiquês*...Caso contrário, não se sinta incomodado pelas fórmulas, apenas foque na ideia principal; algumas analogias ao mundo real que procurei estabelecer irão ajudá-lo a entender também.

1. Modelo computacional de um neurônio

Inspirando-se no funcionamento dos neurônios biológicos do

sistema nervoso dos animais, estabeleceu-se na área da Inteligência Artificial um modelo computacional de um neurônio conforme ilustrado a seguir:



Modelo computacional de um neurônio

Os sinais da entrada no neurônio são representados pelo vetor ***x*** = [*x1, x2, x3, …, xN*], podendo corresponder aos pixels de uma imagem, por exemplo. Ao chegarem ao neurônio, são multiplicados pelos respectivos pesos sinápticos, que são os elementos do vetor ***w*** = [*w1, w2, w3, …, wN*], gerando o valor *z*, comumente denominado potencial de ativação, de acordo com a expressão:

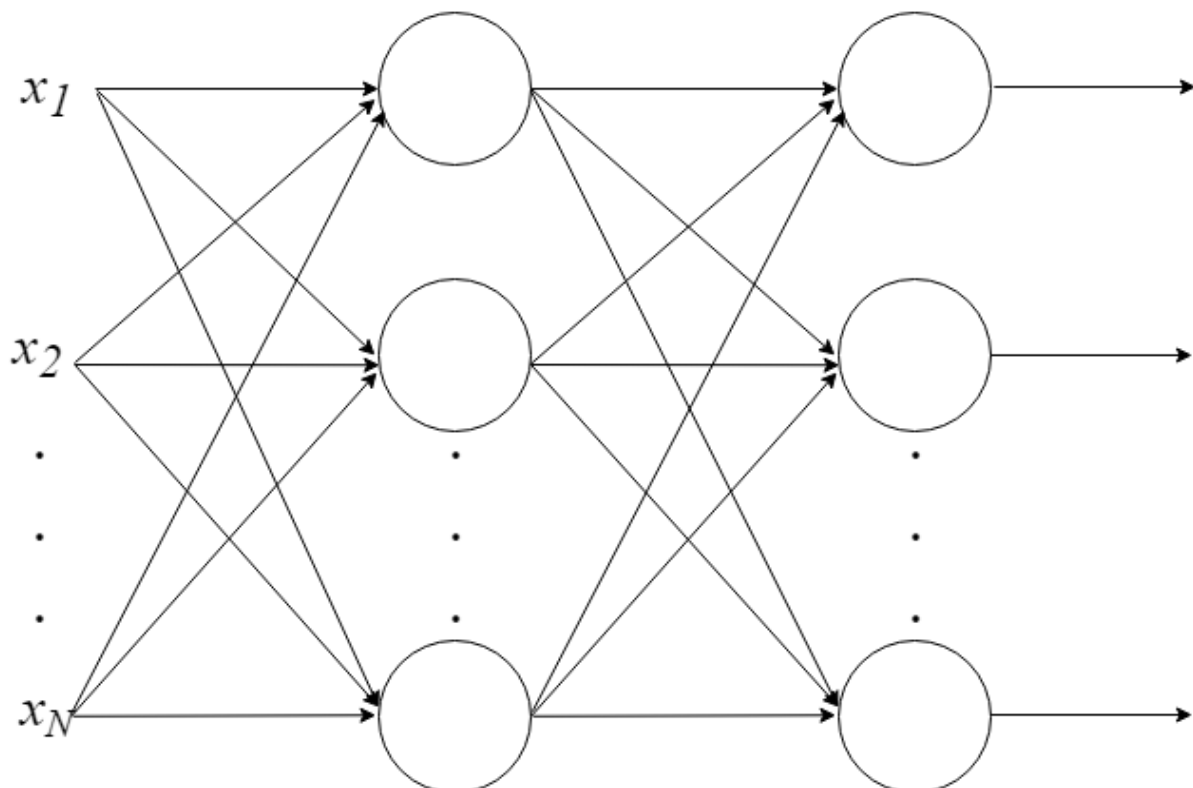
$$z = \sum_{i=1}^{N} x_i w_i + b$$

O termo adicional *b* provê um grau de liberdade a mais, que não é afetado pela entrada nessa expressão, correspondendo tipicamente ao "*bias*" (viés). O valor *z* passa então por uma função matemática de ativação *σ,* com a característica de ser não linear, responsável

por limitar tal valor a um certo intervalo, produzindo o valor final de saída *y* do neurônio. Algumas funções de ativação usadas são a degrau, sigmoide, tangente hiperbólica, softmax e ReLU (*Rectified Linear Unit*).

2. Combinando neurônios em camadas

Com apenas um neurônio não se pode fazer muita coisa, mas podemos combiná-los em uma estrutura em camadas, cada uma com número diferente de neurônios, formando uma rede neural denominada Perceptron Multicamadas ("*Multi Layer Perceptron* —MLP**"**). O vetor de valores de entrada ***x*** passa pela camada inicial, cujos valores de saída são ligados às entradas da camada seguinte, e assim por diante, até a rede fornecer como resultado os valores de saída da última camada. Pode-se arranjar a rede em várias camadas, tornando-a profunda e capaz de aprender relações cada vez mais complexas.



Neurônios combinados formando um rede. Cada círculo representa um neurônio como aquele descrito anteriormente.

3. Treinamento de um MLP

Para que uma rede dessas funcione, é preciso treiná-la. É como ensinar a uma criança o beabá. O treinamento de uma rede MLP insere-se no contexto de aprendizado de máquina supervisionado, em que cada amostra de dados utilizada apresenta um rótulo informando a que classificação ela se encaixa. Por exemplo, uma imagem de um cachorro contém um rótulo informando que aquilo é um cachorro. Assim, a ideia geral é fazer com que a rede aprenda os padrões referentes a cada tipo de coisa (cada *classe*), assim, quando uma amostra desconhecida for fornecida à rede, ela seja capaz de estabelecer a qual classe tal amostra pertence. Como isso pode ser feito?

Backpropagation

> *"Não é errando que se aprende?"*

A ideia do algoritmo backpropagation é, com base no cálculo do **erro** ocorrido na camada de saída da rede neural, recalcular o valor dos pesos do vetor ***w*** da camada última camada de neurônios e assim proceder para as camadas anteriores, de trás para a frente, ou seja, atualizar todos os pesos ***w*** das camadas a partir da última até atingir a camada de entrada da rede, para isso realizando a retropropagação o erro obtido pela rede. Em outras palavras, calcula-se o erro entre o que a rede achou que era e o que de fato era (era um gato e ela achou que era um cachorro—temos aí um *erro!),* então recalculamos o valor de todos os pesos, começando da última camada e indo até a primeira, sempre tendo em vista diminuir esse erro.

Traduzindo essa ideia para o *matematiquês*, o backpropagation consiste nas seguintes etapas:

- Inicializar todos os pesos da rede com pequenos valores **aleatórios**.
- Fornecer dados de entrada à rede e calcular o valor da **função** de erro obtida, ao comparar com o valor de saída esperado. Lembre-se de que como o aprendizado é supervisionado, já se sabe de antemão qual deveria ser a resposta correta. É importante que a função de erro seja **diferenciável**.
- Na tentativa de minimizar o valor da função de erro, calculam-se os valores dos **gradientes** para cada peso da rede. Do **Cálculo**, sabemos que o vetor gradiente fornece a direção de maior **crescimento** de uma função; aqui, como queremos caminhar com os pesos na direção de maior **decréscimo** da função de erro, basta tomarmos o sentido contrário ao do gradiente e...voilà! Já temos um excelente caminho por onde andar.
- Uma vez que temos o vetor gradiente calculado, atualizamos cada peso de modo **iterativo**, sempre recalculando os gradientes em cada passo de iteração, até o erro diminuir e chegar abaixo de algum limiar preestabelecido, ou o número de iterações atingir um valor máximo, quando enfim o algoritmo termina e a rede está treinada.

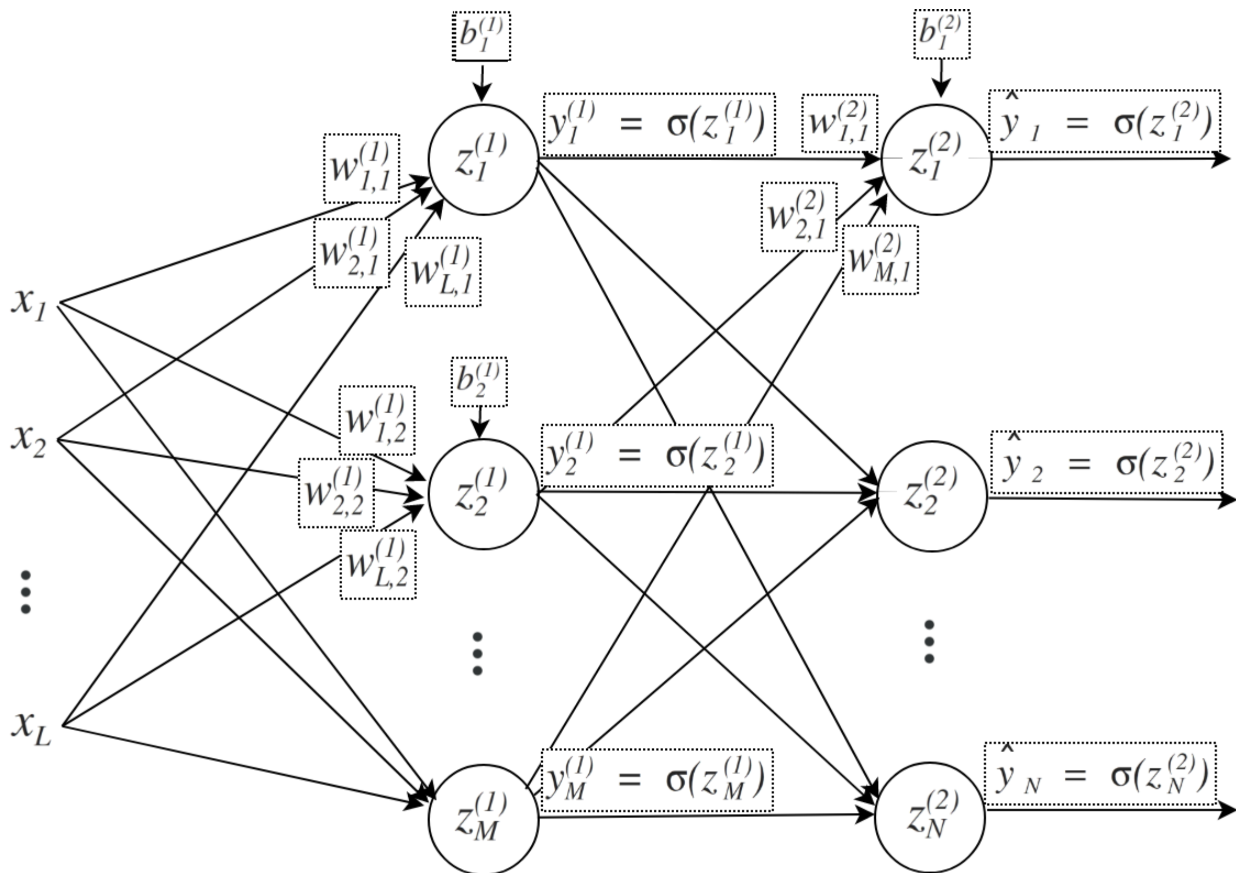
Assim, a fórmula geral de atualização dos pesos na iteração fica:

$$w \leftarrow w - \eta \frac{\partial E}{\partial w}$$

Fórmula geral de atualização dos pesos da rede neural

Ou seja, o valor do peso na iteração atual será o valor do peso na iteração anterior, corrigido de valor proporcional ao gradiente. O sinal negativo indica que estamos indo na direção contrária à do gradiente, conforme mencionado. O parâmetro η representa a taxa de aprendizado da rede neural, controlando a tamanho do passo que tomamos na correção do peso.
Fazendo uma analogia, imagine que você está numa região montanhosa e escura, desejando descer o mais rápido possível à procura de um vale, na esperança de encontrar um lago de águas límpidas. É possível andar em várias direções, mas você tenta achar a melhor ao sentir o solo ao redor e tomar aquela com maior declive. Em termos matemáticos, você está indo na direção contrária à do vetor gradiente. Considere também que você pode controlar o tamanho do passo que pode dar, mas note que passos muito largos podem fazê-lo passar do vale, caindo na outra montanha mais adiante, e outro passo largo o trará de volta à primeira montanha, fazendo-o pular pra lá e pra cá sem nunca alcançar o vale; por outro lado, passos muito curtos o farão demorar muito a descer, podendo morrer de sede do meio do caminho.

Voltando ao *matematiquês*, o conceito-chave da equação anterior é o cálculo da expressão ***∂E /∂w,*** consistindo em computar as **derivadas parciais** da função de erro ***E*** em relação a cada peso do vetor ***w***. Para nosso auxílio, vamos considerar a figura a seguir, que ilustra uma rede MLP com duas camadas e servirá de base para a explicação do backpropagation. Uma conexão entre um neurônio ***j*** e um neurônio ***i*** da camada seguinte possui peso ***w[j,i]***. Repare que os números sobrescritos, entre parênteses, indicam o número da camada à qual a variável em questão pertence, podendo, neste exemplo, valer (1) ou (2).



Rede MLP para ilustrar a explicação do backpropagation. Os pesos dos neurônios inferiores foram omitidos para não tornar a figura visualmente poluída.

Sinta-se à vontade para voltar à figura anterior conforme for acompanhando os passos a seguir. Sendo ***y*** a saída esperada e ***ŷ*** a saída obtida pela rede, definimos a função de erro como sendo:

$$E(y, \hat{y}) = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Ou seja, aqui estamos simplesmente calculando a somatória dos quadrados das diferenças entre os elementos dos dois vetores. Agora, calculamos a derivada parcial do erro em relação à camada de saída, ***ŷ*:**

$$\frac{\partial E}{\partial \hat{y}_k} = \frac{\partial}{\partial \hat{y}_k}\left(\sum_{i=1}^{N}(y_i - \hat{y}_i)^2\right) = \frac{\partial}{\partial \hat{y}_k}(y_k - \hat{y}_k)^2 = -2(y_k - \hat{y}_k)$$

Seguindo a estrutura da rede para a esquerda, prosseguimos no cálculo da derivada do erro em relação ao potencial de ativação ***z*** da camada (2), usando a **Regra da Cadeia**.

$$\frac{\partial E}{\partial z_i^{(2)}} = \frac{\partial E}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial z_i^{(2)}} = -2(y_i - \hat{y}_i)\sigma'(z_i^{(2)})$$

Note que ***σ'*** é a derivada da função de ativação do neurônio. Por exemplo, no caso de ser a função sigmoide, sua derivada vale $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

Este valor é o gradiente local em relação ao *i*-ésimo neurônio da camada (2) e, para não tornar as fórmulas excessivamente extensas, será simplesmente indicado como ***δ***:

$$\frac{\partial E}{\partial z_i^{(2)}} = \delta_i^{(2)}$$

Finalmente, usando a Regra da Cadeia mais uma vez, computamos a derivada parcial do erro em função de um peso ***w[j,i]*** da camada (2):

$$\frac{\partial E}{\partial w_{j,i}^{(2)}} = \frac{\partial E}{\partial z_i^{(2)}}\frac{\partial z_i^{(2)}}{\partial w_{j,i}^{(2)}} = \delta_i^{(2)}\frac{\partial}{\partial w_{j,i}^{(2)}}\left(\sum_{k=1}^{M} w_{k,i}^{(2)}\, y_k^{(1)} + b_i^{(2)}\right) = \delta_i^{(2)} y_j^{(1)}$$

O cálculo em relação ao *bias* é análogo, resultando em:

$$\frac{\partial E}{\partial b_i^{(2)}} = \delta_i^{(2)}$$

Com esses resultados em mãos, é possível aplicar a fórmula geral de atualização dos pesos dos neurônios da última camada:

$$w_{j,i}^{(2)} \leftarrow w_{j,i}^{(2)} - \eta \delta_i^{(2)} y_j^{(1)}$$

E a fórmula de atualização do *bias:*

$$b_i^{(2)} \leftarrow b_i^{(2)} - \eta \delta_i^{(2)}$$

Pronto, já temos a fórmula mágica para atualizar os pesos dos neurônios da última camada. Agora temos que calcular esta expressão para os pesos dos neurônios da camada anterior; logo, precisamos calcular a derivada parcial em relação à saída **y[i]** da camada (1). A sacada aqui é perceber que tal valor **y[i]** interfere em todos os neurônios da camada seguinte, assim, temos que considerar a somatória dos erros que ele propaga:

$$\frac{\partial E}{\partial y_i^{(1)}} = \sum_{j=1}^{N} \frac{\partial E}{\partial z_j^{(2)}} \frac{\partial z_j^{(2)}}{\partial y_i^{(1)}}$$

Mas:

$$\frac{\partial z_j^{(2)}}{\partial y_i^{(1)}} = \frac{\partial}{\partial y_i^{(1)}} \left(\sum_{k=1}^{M} w_{k,j}^{(2)} y_k^{(1)} + b_j^{(1)} \right) = w_{i,j}^{(2)}$$

Resultando em:

$$\frac{\partial E}{\partial y_i^{(1)}} = \sum_{j=1}^{N} \delta_j^{(2)} w_{i,j}^{(2)}$$

Prosseguindo, temos:

$$\frac{\partial E}{\partial z_i^{(1)}} = \frac{\partial E}{\partial y_i^{(1)}} \frac{\partial y_i^{(1)}}{\partial z_i^{(1)}} = \left(\sum_{j=1}^{N} \delta_j^{(2)} w_{i,j}^{(2)} \right) \sigma'(z_i^{(1)}) = \delta_i^{(1)}$$

Esse ***δ*** a que igualamos o resultado segue a mesma ideia anterior: é o gradiente local em relação ao *i*-ésimo neurônio da camada (1). Finalmente, vem:

$$\frac{\partial E}{\partial w_{j,i}^{(1)}} = \frac{\partial E}{\partial z_i^{(1)}} \frac{\partial z_i^{(1)}}{\partial w_{j,i}^{(1)}} = \delta_i^{(1)} \frac{\partial}{\partial w_{j,i}^{(1)}} \left(\sum_{k=1}^{L} w_{k,i}^{(1)} x_k + b_i^{(1)} \right) = \delta_i^{(1)} x_j$$

Substituindo na fórmula de atualização dos pesos:

$$w_{j,i}^{(1)} \leftarrow w_{j,i}^{(1)} - \eta \delta_i^{(1)} x_j$$

Novamente, as fórmulas para o *bias* são análogas e ficam como exercício.

É isso! Caso existissem mais camadas na rede, o procedimento continuaria, seguindo sempre esse mesmo padrão de calcular as derivadas parciais, retropropagando os erros até a camada de entrada da rede.

O backpropagation é um algoritmo elegante e engenhoso. Os atuais modelos *deep learning* como Redes Neurais Convolucionais,

embora mais refinados que o MLP, têm se mostrado muito superiores em tarefas como classificação de imagens e também utilizam como método de aprendizado o backpropagation, assim como as chamadas Redes Neurais Recorrentes, em processamento de linguagem natural, também fazem uso desse algoritmo. O mais incrível é que tais modelos conseguem encontrar padrões inobserváveis e obscuros para nós, humanos, o que é fascinante e permite considerar que em breve estaremos contando com a ajuda do *deep learning* para resolver muitos dos principais problemas que afligem a humanidade, como a cura do câncer.