Aula 6

1 Inspiração Biológica

Sabemos que o cérebro humano é capaz de processar diversos tipos de dados de uma maneira dinâmica e complexa, podendo se adaptar a vários cenários. Pensando nisso, porque não desenvolvemos um sistema de machine learning (ML) capaz de também ter essas mesmas características? Foi com esse pensamento que surgiu a ideia das primeiras redes neurais.

Como destacaremos na seção seguinte, houve uma inspiração profunda na neurologia por parte dos cientistas da computação, refletida na nomenclatura 'rede neural artificial', e na estruturação desse modelo de Machine Learning. Das ideias do psicólogo Donald Hebb sobre a conexão entre neurônios relacionados se fortalecer, surgiu o perceptron, que, encadeado e levemente modificado, tornou-se a base para o atual Deep Learning.

Contudo, é de extrema importância destacar que os processos que acontecem em uma rede neural são diferentes do real processo que acontece no cérebro humano. Tal qual um avião é inspirado nos pássaros, mas distinto deles, ANNs (Artificial Neural Networks) não são um cérebro humano. O que dá a esse tipo de sistema esse nome é a sua inspiração em replicar alguns fenômenos que acontecem no aprendizado humano, visando conseguir a versatilidade adaptativa que é vista nos sistemas biológicos. Um exemplo disso é o neurônio, que, tanto no cérebro quanto em uma rede neural, é uma unidade de processamento básica, sendo que, claramente, o neurônio de uma rede neural é adaptado para sistemas computacionais e os tipos de processos que podem ocorrer neles (elétricos, e não eletroquímicos).

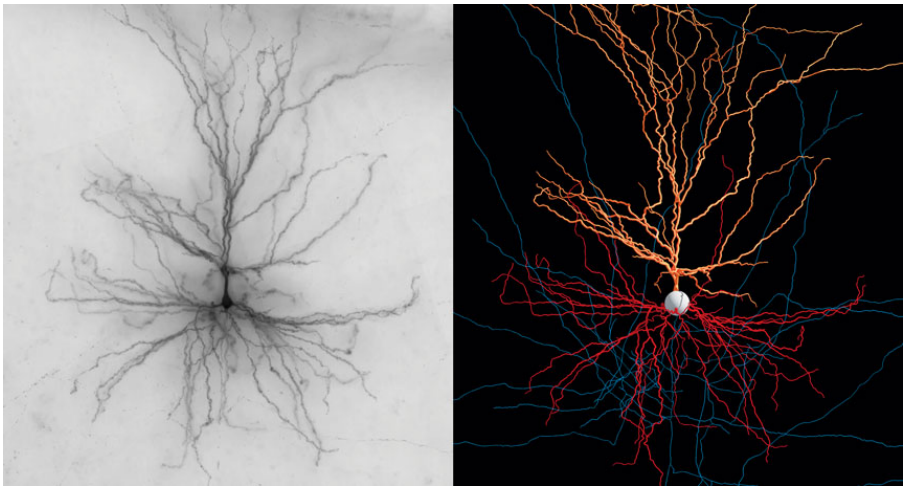


Figura 1: O neurônio humano: tão similar quanto distinto do artificial

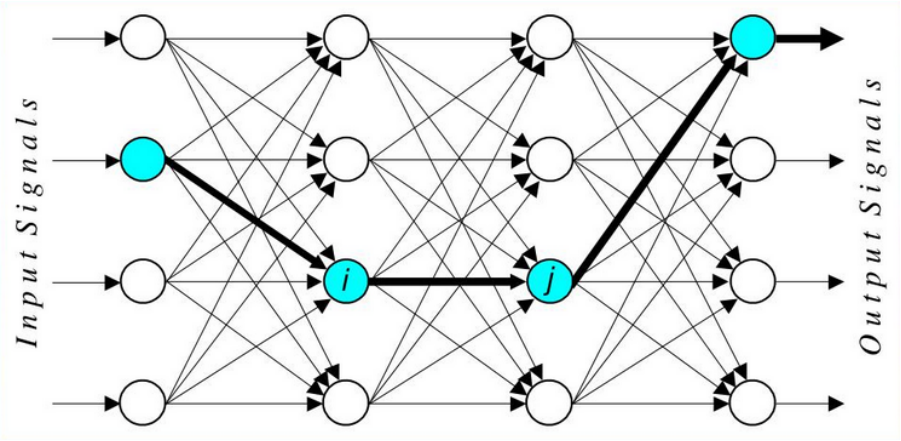


Figura 2: Modelo idealizado por Hebb do funcionamento neuronal humano

1.1 Tipos

Existem vários tipos de redes neurais, cada uma com uma arquitetura e um propósito diferentes. Alguns exemplos são:

- **Multi-Layer Perceptron (MLP):**
  Rede neural mais simples de vários neurônios totalmente conectados entre as camadas. Essa arquitetura é utilizada para resolver diversos tipos de problemas, sendo bem versátil e simples de implementar; por exemplo, nessa aula, exibiremos uma implementação clássica de MLP: o reconhecimento de dígitos numéricos.

- **Convolutional Neural Networks (CNN):**
  Esse tipo de arquitetura é usada principalmente para analisar imagens, utilizando o processo matemático da convolução para conseguir extrair features dos dados. As CNNs são muito utilizadas, por exemplo, na área médica, para classificação e identificação de comorbidades.

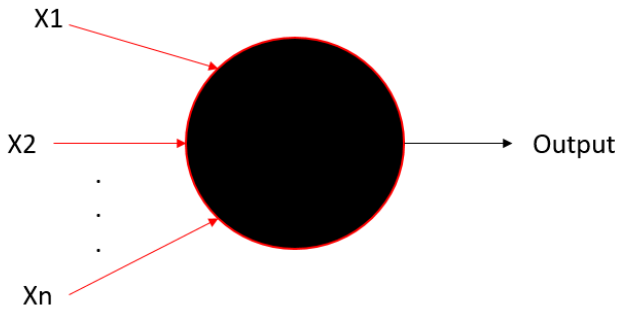
Figura 3: aplicação de CNN: identificação (básica) de targets; source = "Tell Me Where to Look: Guided Attention Inference Network"

- **Recurrent Neural Networks (RNN):**
  Já esse tipo de redes neurais é utilizado principalmente quando estamos falando de sequências, como em um texto ou um áudio, em que o input anterior impacta no próximo input analisado. A ferramenta de auto-complete do Google, por exemplo, é uma RNN.

1.2 Neurônio Simples

Nesse sentido, o que seria um neurônio em uma rede neural? Lógicamente ele não será como os nossos, mas o que seria o axônio e o dendrito no software? Vamos imaginar o seguinte cenário, como em outros casos de ML, temos as nossas features; dessa forma, o nosso neurônio deve ser capaz de conseguir processar essas features (input) e devolver um certo resultado (output).

Perceba que o seguinte modelo satisfaz precisamente essas necessidades:



Dessa forma, dado um input, que pode ser composto de várias features, o nosso neurônio irá produzir um único output, seguindo a seguinte fórmula:

$$\text{out} = \gamma(Wx + b) \tag{1}$$

Onde W é um vetor de pesos de mesma dimensões que o vetor de input x, b é um treshold arbitrário para a ativação, e γ, uma activation function genérica. Note que nosso neurônio artificial se conecta com muitos outros na rede, com diferentes relevâncias para cada conexão x_n (dadas pelos respectivos pesos, w_n), e também "ativa"tal qual o biológico devido a γ (com a diferença dessa ativação não ser binária, no caso da máquina). Note a similaridade entre esse modelo descrito e o do Perceptron, apresentado na aula 1, sendo a única diferença a função de ativação.

2 Redes Neurais: Multi-Layer-Perceptrons

2.1 Estrutura multi-camadas

Apresentaremos agora o tipo mais simples de rede neural: os MLPs. Seu nome deriva do fato de cada neurônio nas layers ter sido inspirado nos nossos primeiros amigos do curso: os Perceptrons!

MLPs são como ogros, ou seja, contém camadas. A separação básica envolve uma camada de inputs (cada neurônio recebe uma feature), uma de output (que varia muito com o que você deseja que ela aprenda), e diversas camadas intermediárias, as chamadas 'hidden layers'. A esperança é de que, durante diversos treinamentos, os pesos entre os neurônios se ajustem de modo a consolidar um entendimento dos padrões no dataset, reduzindo uma Loss Function pré-estabelecida. Note que isso não é nada diferente do que já fazíamos antes nos outros modelos (SVM, perceptron, regressão linear); é apenas mais um sabor de Machine Learning.

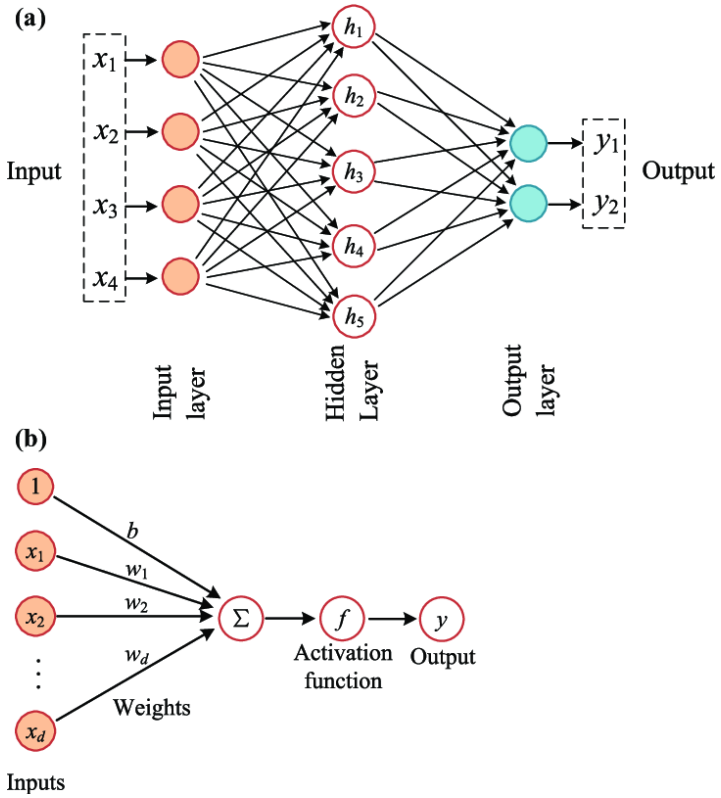


Figura 4: Estrutura básica de um MLP; créditos a Jiang Kejie

2.2 Quebra da linearidade: ativação

Para um neurônio n, sua ativação é dada por:

$$\text{out}_n = \gamma(W_n \cdot \text{out}_{n-1} + b_n)$$

Você pode estar se perguntando qual a função prática dessa função de ativação. Repare: A função $Wx + b$ é linear, e, sendo assim, modelos que somente a utilizam tornam-se incapazes de detectar no dataset padrões não lineares (lembrem de toda a ideia por trás das features transformations em SVMs!), tornando-a bem 'ruinzinha'.

Mesmo caso empilhemos várias camadas de neurônios s/ativação, a limitação linear não desaparece. Imagine, por exemplo, uma rede neural deles com várias camadas, ou seja:

$$\begin{aligned} \text{Camada 1:} \quad & z_1 = W_1 x + b_1 \\ \text{Camada 2:} \quad & z_2 = W_2 z_1 + b_2 \\ & \vdots \end{aligned}$$

Se todas essas forem funções lineares, então a composição de todas elas também será uma única função linear isto é:

$$z = W_n(...W_2(W_1 x + b_1) + b_2...) + b_n = W'x + b' \tag{2}$$

Não importa quantas camadas você tenha, a combinação entre elas é como se fosse uma única camada linear! Sendo assim, para resolver o problema da linearidade iremos, desta vez, ao invés de utilizar Funções Kernel, utilizar funções de ativação; pense nelas como maneiras práticas de introduzir não linearidade na rede neural.

Seguem alguns exemplos:

- **ReLU (Rectified Linear Unit):** $\sigma(z) = \max(0, z)$
- **Sigmoid:** $\sigma(z) = \dfrac{1}{1 + e^{-z}}$
- **tanh:** $\sigma(z) = \tanh(z)$

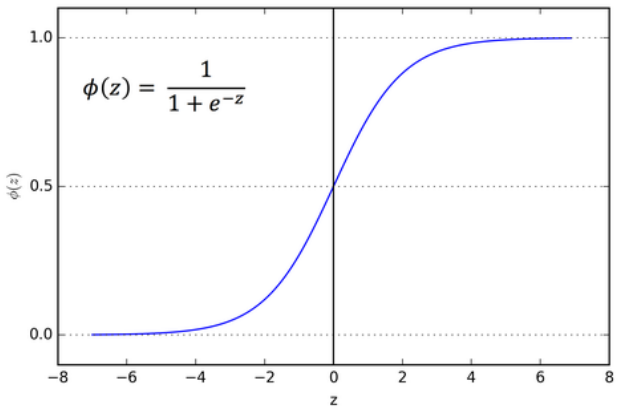


Figura 5: Sigmóide (σ)

2.3 Vantagens das Redes Neurais

Por qual motivo o modelo que descrevemos até agora é um dos mais famosos do mundo? Bem, há várias vantagens que as redes neurais apresentam, tornando-as muito queridinhas para os cientistas de dados da atualidade. De forma simplificada, algumas delas são:

- **Capacidade para Manifold Learning:** em ML, frequentemente deseja-se encontrar um subespaço com algumas variações mais simples em um espaço vetorial gigantesco (manifoldo). Pense em uma imagem com 1024 pixels, sendo cada pixel uma feature: deseja-se encontrar (para detecção facial) um manifoldo em $\mathbb{R}^{1024}$ (o espaço de todas as imagens possíveis nessas dimensões) que detecte traços como nariz, olhos, e etc.

- **"Maldição"da Dimensionalidade**: quanto maior a dimensionalidade, maior o número de dados para que os modelos detectem padrões, sendo que a razão $\frac{\text{dados}}{\text{dimensão}}$ cresce exponencialmente. As redes neurais conseguem reduzir essa complexidade significativamente, apesar de ainda sofrerem com ela.

- **Mais priors:** um prior é algo que assumimos sobre a função que queremos estimar no feature space do dataset. Redes Neurais, por serem o modelo de ML que mais permite integrar priors no aprendizado, facilita a detecção de padrões pela máquina (na próxima aula, isso ficará mais claro com 'pooling' nas CNNs)

3 Como treinar uma rede neural

Backpropagation (retropopagação) é o **algoritmo que calcula os gradientes da função de perda com relação a todos os parâmetros da rede** (pesos e biases), aplicando a **regra da cadeia (chain rule)** do cálculo diferencial.

3.1 O que exatamente o backpropagation resolve?

O objetivo do backpropagation é **ajustar os parâmetros da rede neural** (os pesos e biases de cada camada) para **minimizar a função de perda**.

Para isso, precisamos saber **como cada peso e bias influencia a perda final**. Isso é representado pelos **gradientes parciais**:

$$\frac{\partial L}{\partial w_{ij}^{(l)}}, \quad \frac{\partial L}{\partial b_j^{(l)}}$$

Onde:

- L: função de perda
- $w_{ij}^{(l)}$: peso da camada l, que conecta o neurônio i da camada anterior ao neurônio j da camada atual
- $b_j^{(l)}$: bias do neurônio j da camada l

Esses valores nos dizem:

> "Se eu alterar ligeiramente esse peso, quanto a perda muda?"

3.2 Como funciona o algoritmo?

O algoritmo de backpropagation segue 2 fases principais: forward pass e backward pass.

3.2.1 Forward Pass (passagem para frente)

A primeira etapa do backpropagation pode ser descrita pelos seguintes passos:

- Computamos a saída da rede camada por camada (inclusive a loss final).
- Salvamos os valores intermediários: entradas $x^{(l)}$, ativações $a^{(l)}$, pré-ativações $z^{(l)}$, conforme segue.

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = \sigma(z^{(l)})$$

3.2.2 Backward Pass (retropopagação)

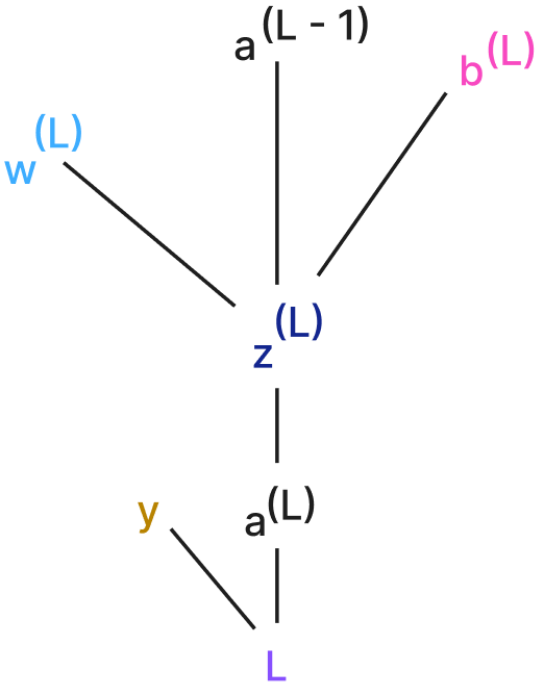
Vamos começar com um exemplo simples, onde temos 3 neurônios conectados em sequência:

Chamaremos o último neurônio de $a^{(L)}$, o penúltimo de $a^{(L-1)}$ e assim por diante.

A primeira coisa que devemos fazer é calcular o erro:

$$L = (a^{(L)} - y)^2$$

A questão é que temos que lembrar que $a^{(L)}$ é uma função que pode ser quebrada em:



Sendo assim, para calcularmos a derivada de L em relação a $w^{(L)}$, $a^{(L-1)}$, $b^{(L)}$ teremos que utilizar a regra da cadeia que é dada por:

$$\frac{\partial L}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial L}{\partial a^{(L)}}$$

Temos que

1. $L = (a^{(L)} - y)^2$, logo: $\frac{\partial L}{\partial a^{(L)}} = 2(a^{(L)} - y)$
2. $a^{(l)} = \sigma(z^{(l)})$, logo: $\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(l)})$
3. $z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}, logo: \frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$

4. $\frac{\partial L}{\partial w^{(L)}} = a^{(L-1)}.\sigma'(z^{(l)}).2(a^{(L)} - y)$

$$\frac{\partial L}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial L}{\partial a^{(L)}}$$

Temos que:

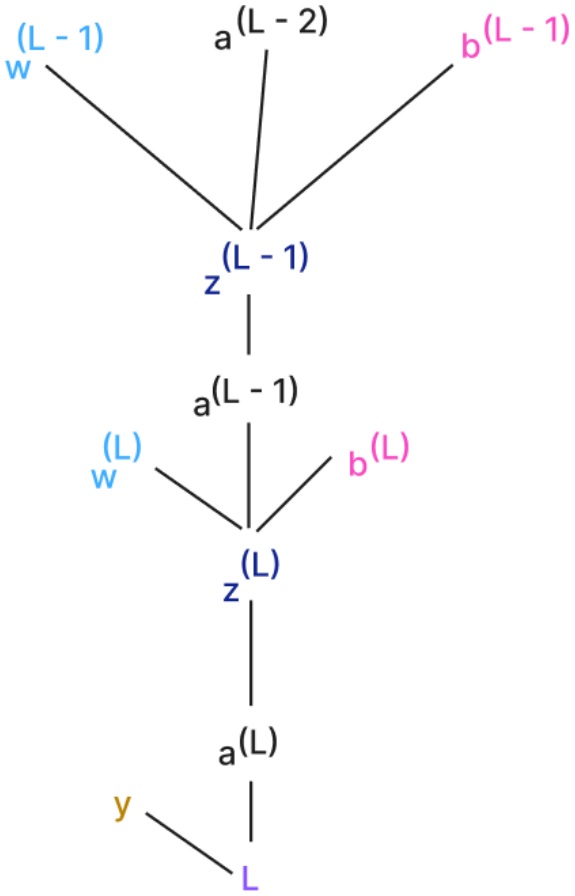
1. $z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}, logo : \frac{\partial z^{(L)}}{\partial b^{(L)}} = 1$
2. $\frac{\partial L}{\partial w^{(L)}} = 1.\sigma'(z^{(l)}).2(a^{(L)} - y)$

$$\frac{\partial L}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial L}{\partial a^{(L)}}$$

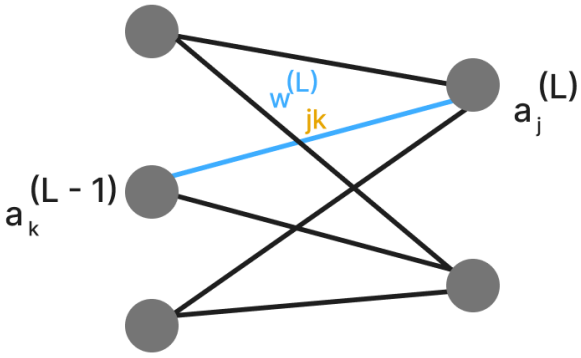
Temos que:

1. $z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}, logo : \frac{\partial z^{(L)}}{\partial a^{(L-1)}} = W^{(l)}$
2. $\frac{\partial L}{\partial w^{(L)}} = W^{(L)}.\sigma'(z^{(l)}).2(a^{(L)} - y)$

Depois você irá repetir o processo para o resto das camadas:



Mas e quando temos uma rede neural mais complicada? Onde temos mais de um neurônio em uma dada camada.



Vamos ter então:

$$L = \sum_{j=0}^{n} (a_j^{(L)} - y_j)^2$$

Sendo assim, o nosso cálculo passará a ser:

$$\frac{\partial L}{\partial W_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial W_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial L}{\partial a_j^{(L)}}$$

$$\frac{\partial L}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial L}{\partial a_j^{(L)}}$$

Sendo assim, teremos:

$$\frac{\partial \mathcal{C}}{\partial w_{jk}^{(l)}} = a_k^{(l-1)} \, \sigma' \left(z_j^{(l)} \right) \frac{\partial L}{\partial a_j^{(l)}}$$

$$\frac{\partial \mathcal{C}}{\partial a_j^{(l)}} \begin{cases} \displaystyle\sum_{j=0}^{n} w_{jk}^{(l+1)} \, \sigma' \left(z_j^{(l+1)} \right) \frac{\partial L}{\partial a_j^{(l+1)}} \\ \text{ou} \quad 2 \left(a_j^{(L)} - y_j \right) \end{cases} \tag{3}$$

3.2.3 Atualização (gradient descent)

A partir dos gradientes calculados, podemos então atualizar os parâmetros da rede de modo a diminuir a loss. Para isso, utilizaremos a ideia do gradiente descendente com uma learning rate η (normalmente um valor entre 10^{-3} e 10^{-5}). Por exemplo, para o peso w_{11}, teremos:

$$w_{11} := w_{11} - \eta \cdot \frac{\partial L}{\partial w_{11}}$$

A partir desses processos é que uma rede neural aprende e é capaz de generalizar situações para diferentes problemas.