iscte
TECNOLOGIAS
E ARQUITETURA

# APVC

Redes Neuronais

# Sumário

- Redes Neuronais
  - Conceitos
  - Treino de uma rede neuronal
  - Otimização: *gradient descent*
  - Medidas de desempenho
  - Implementação no Tensorflow
  - Exemplos

iscte
TECNOLOGIAS
E ARQUITETURA

# Redes Neuronais

# Aprendizagem automática (*machine learning*)

***Machine learning*** *is a branch of artificial intelligence and computer science which focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy.*

*IBM*

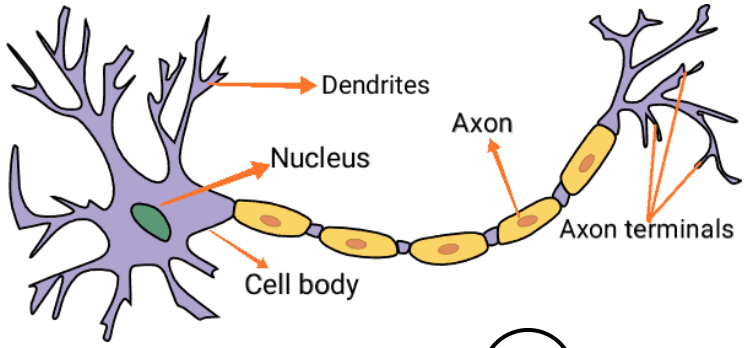
***Machine learning (ML)*** *is a field of study in artificial intelligence concerned with the development and study of statistical algorithms that can learn from data and generalize to unseen data, and thus perform tasks without explicit instructions.*

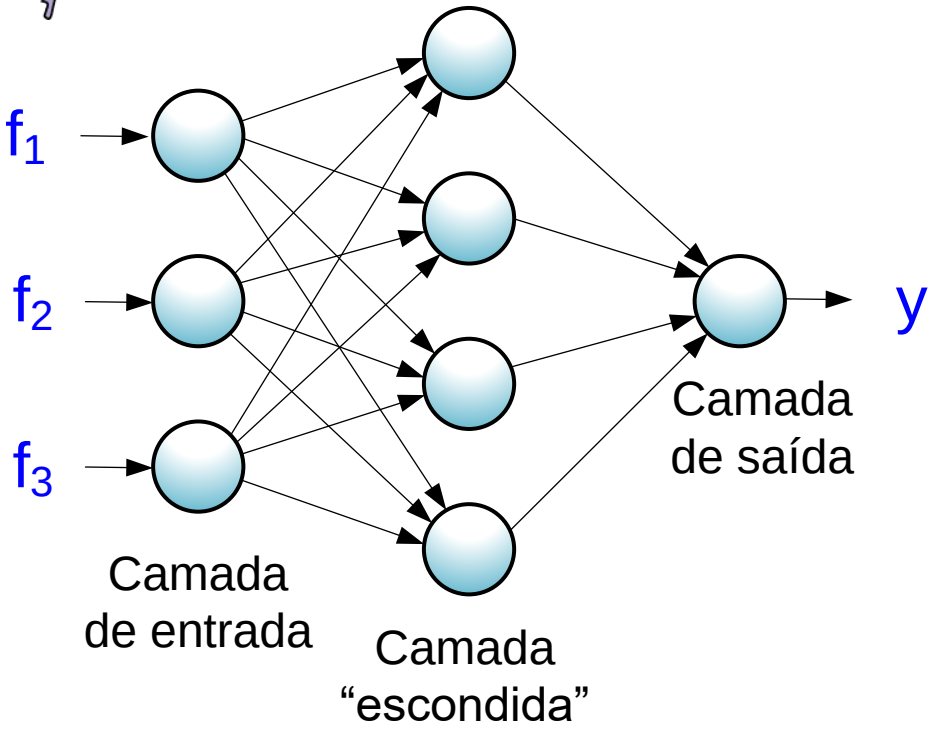
*Wikipedia*

- Tipos de aprendizagem automática:
  - **Supervisionada** – o algoritmo aprende com base em exemplos cujo resultado é conhecido – é neste género de aprendizagem que se encaixam as redes neuronais
  - **Por reforço** – o algoritmo aprende com base em recompensas e penalizações que resultam das ações ou decisões que vai tomando
  - **Não supervisionada** – o algoritmo aprende uma representação interna de um conjunto de dados
  - …

# Redes neuronais “clássicas”

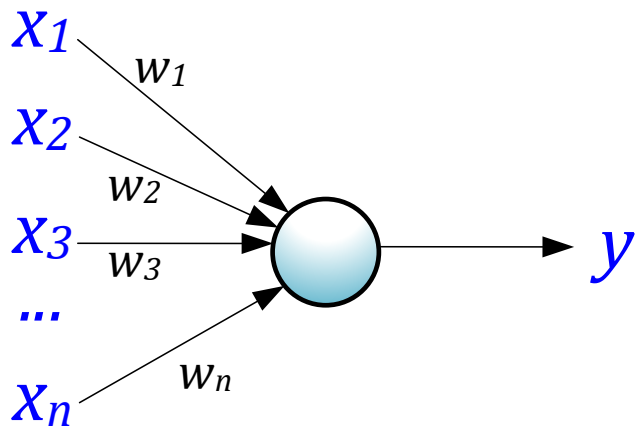
- Inspiradas na biologia
  - No funcionamento do cérebro de mamíferos
- Remontam ao final dos 50’s mas só começaram a ser mais usadas a partir dos 80’s
- Compostas por unidades designadas por **neurónios**, organizadas segundo **camadas**
- A rede aceita valores dos atributos (*features*) e produz predições de classes (classificação) ou estimativas de valores (regressão)



# Neurónios (*Perceptron units*)

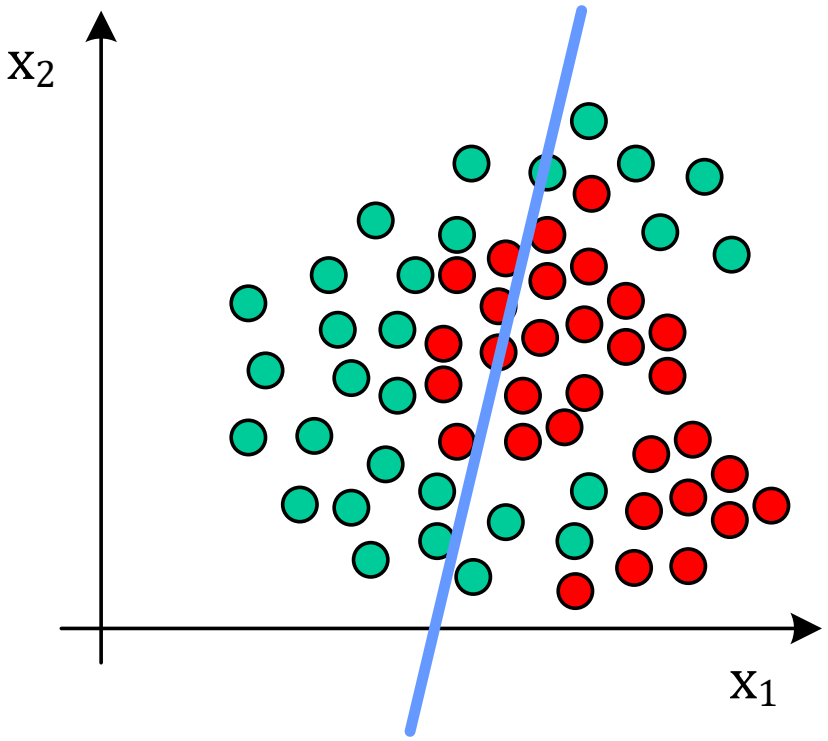
- O cálculo realizado por cada neurónio é uma "soma pesada"
  - Produto interno entre os valores de entrada – $x_i$ – e os pesos associados às ligações – $w_i$



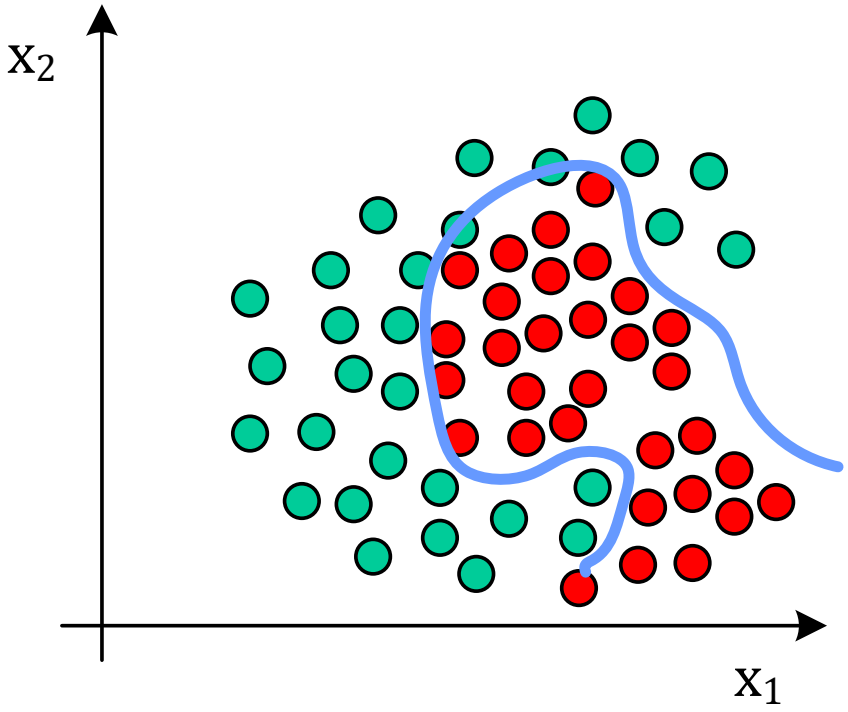

$$y = g\left(w_0 + \sum_{i=1}^{n} w_i \cdot x_i\right)$$

  - $w_0$ é um valor de *offset* designado por *bias*
  - $g(z)$ é uma função de ativação

# Porquê usar uma função de ativação?

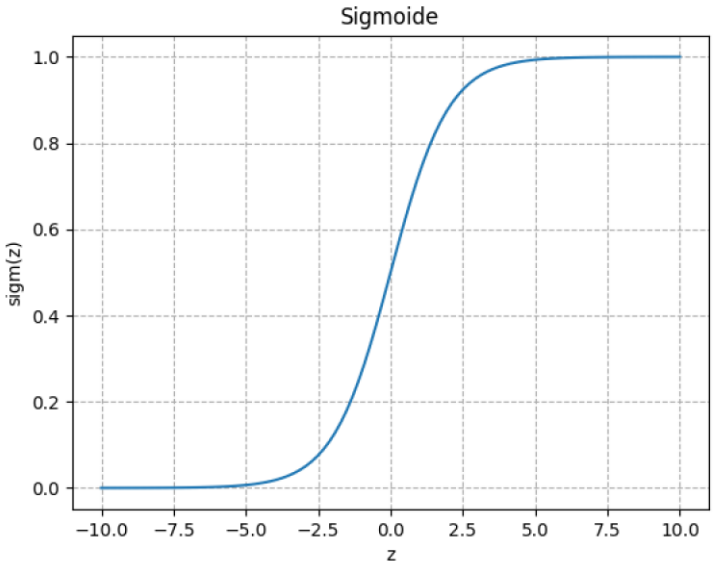


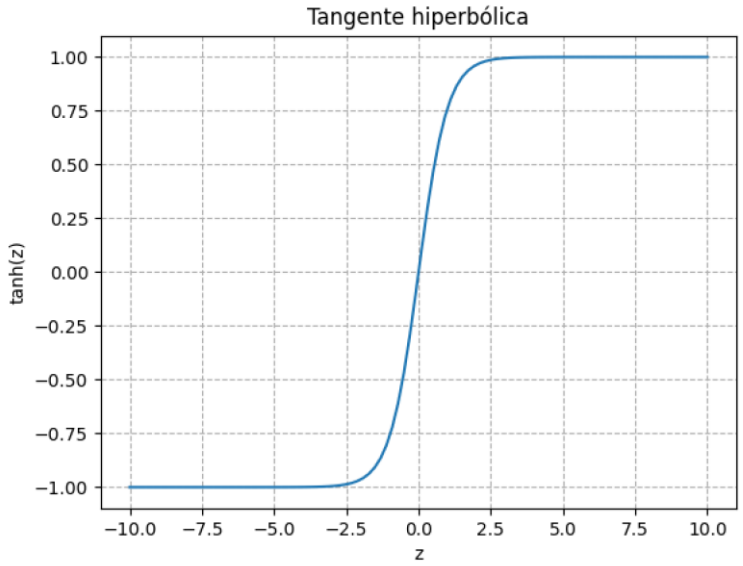

Sem ativação (ou com uma ativação completamente linear), a separação dos dados será sempre linear

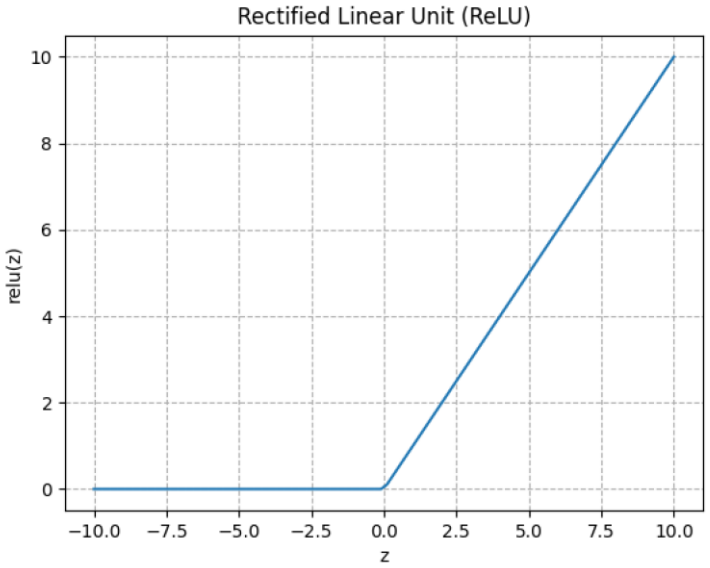



Com funções de ativação conseguem-se separações não lineares, normalmente mais ajustadas aos dados que queremos classificar

# Funções de ativação mais comuns




$$sigm(z) = \frac{1}{1 + e^{-z}}$$

$$sigm'(z) = sigm(z) \times sigm(1 - z)$$

$$tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

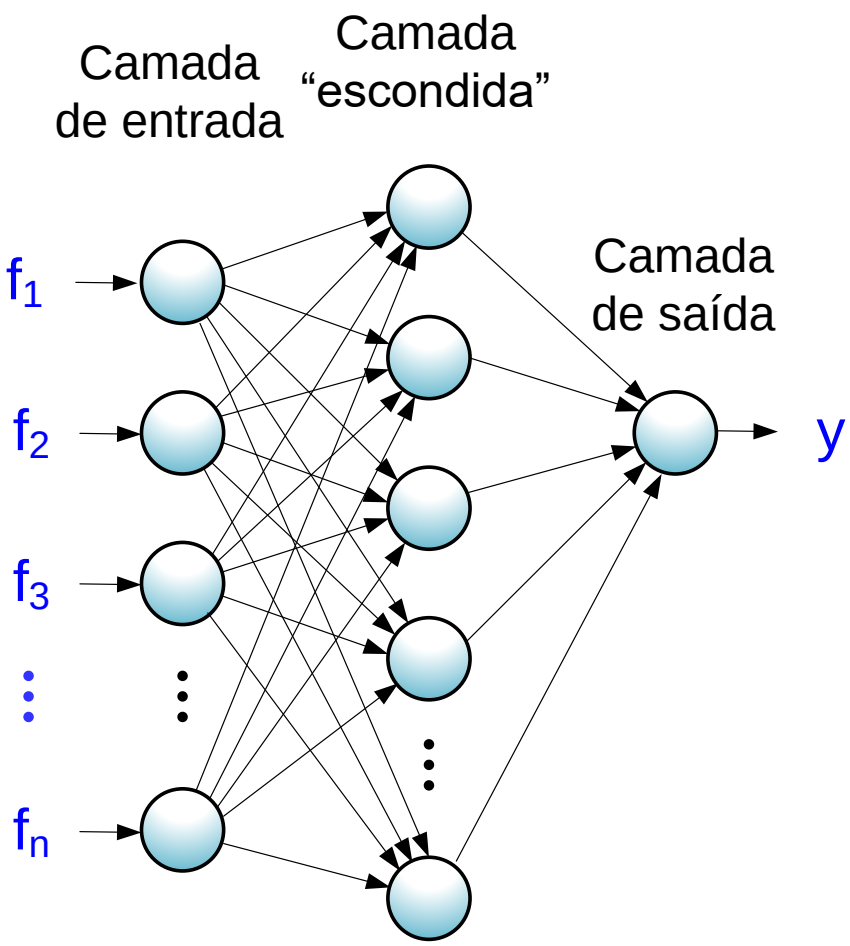
$$tanh'(z) = 1 - tanh(z)^2$$

$$ReLU(z) = \begin{cases} x, se\ x > 0 \\ 0, se\ x \leq 0 \end{cases}$$

$$ReLU'(z) = \begin{cases} 1, se\ x > 0 \\ 0, se\ x \leq 0 \end{cases}$$

**Importante:** de forma a não introduzirem demasiada complexidade no treino das redes, as funções de ativação mais usadas têm derivadas que são simples de tratar matematicamente

# Arquitetura de uma rede neuronal "shallow"

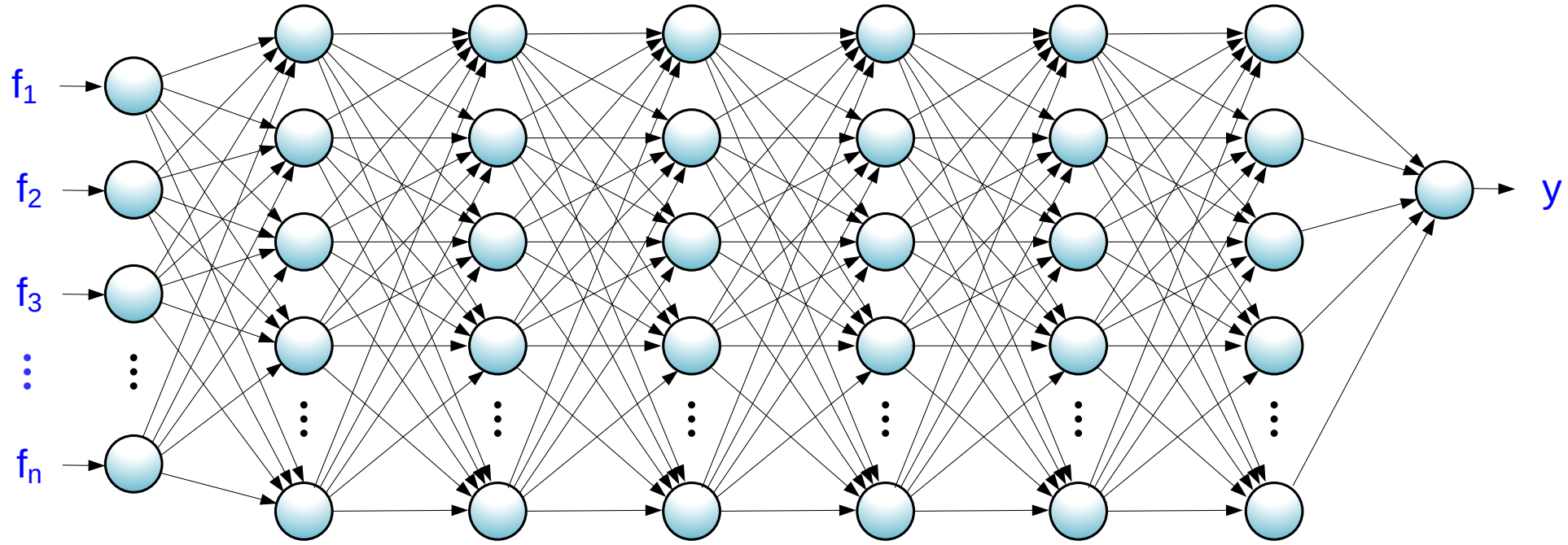



Tipicamente três camadas:

- **Entrada** – camada através da qual os valores dos atributos de cada amostra entram na rede – na prática é uma representação visual para fazer a ligação dos dados de entrada ao resto da rede
- **"Escondida"** – composta por um nº arbitrário de neurónios, cujas entradas estão ligadas a todos os pontos de entrada de dados na rede – é uma camada *Densa* ou *completamente ligada*
- **Saída** – o número de neurónios é fixado consoante o problema:
  - *Estimação de um valor* – um único neurónio, sem função de ativação
  - *Classificação binária* – um único neurónio com uma função de ativação do género sigmoide ou tanh
  - *Classificação multiclasse* – tantos neurónios quanto o número de classes, tipicamente com função de ativação "*softmax*" (valores normalizados que podem ser interpretados como probabilidades)

# *Deep learning*

Habitualmente usa-se o termo *deep learning* para nos referirmos a algoritmos de aprendizagem automática que utilizam redes neuronais com muitas camadas

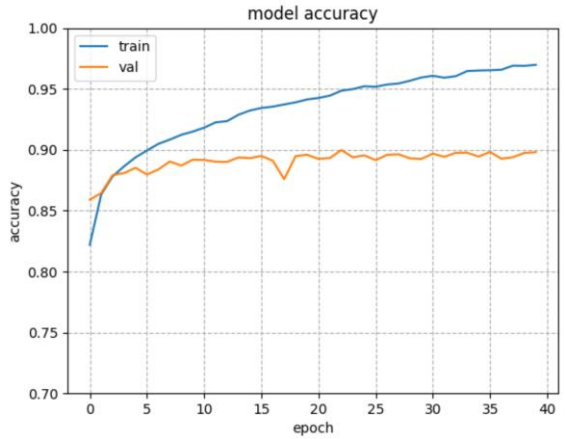


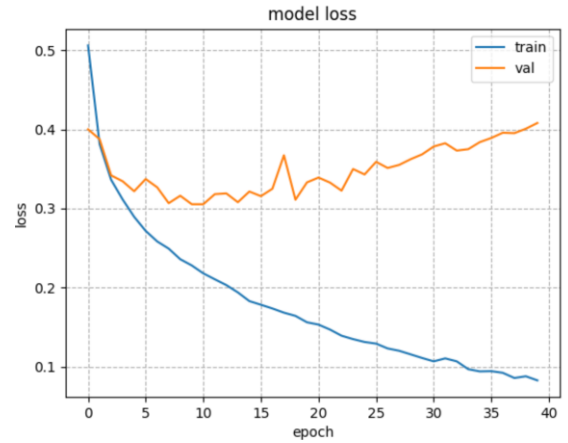

Voltaremos a este tema mais tarde, e com arquiteturas próprias para aplicação a imagens.

# Treino de uma rede neuronal (I)

- Tipicamente os dados são divididos segundo três conjuntos:
  - **Treino** – para calcular e optimizar os pesos da rede
  - **Validação** – para ir avaliando o desempenho durante o treino
  - **Teste** – para medir o desempenho com base em dados novos que não foram usados no treino nem na validação
- A otimização dos pesos é geralmente feita usando o algoritmo de ***back propagation***
  - Usada uma função de perda (***loss***), que quantifica o erro nas predições
  - Procura-se iterativamente o conjunto de pesos que minimiza a função de perda
  - Durante o treino, a propagação do gradiente da função de perda é feita das camadas finais para as iniciais, daí a designação

Acertos (*accuracy*) e perda (*loss*) ao longo do treino de uma rede neuronal



# Funções de perda mais comuns

- *Mean square error* – usada em regressão

$$L_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

$N$ – Nº de amostras
$C$ – Nº de classes (quando aplicável)

$y_i$ – Valor "verdadeiro" da amostra
$\hat{y}_i$ – Valor predito pela rede

- *Binary cross-entropy* – usada em classificação binária

$$L_{BCE} = -\sum_{i=1}^{N} y_i \times \log(\hat{y}_i) + (1 - y_i) \times \log(1 - \hat{y}_i)$$

- *Categorical cross-entropy* – usada em classificação multi-classe

$$L_{CCE} = -\sum_{i=1}^{N}\sum_{j=1}^{C} y_{i,j} \times \log(\hat{y}_{i,j})$$

# Treino de uma rede neuronal (II)

- A função de perda pode ser escrita como uma função dos pesos:

$$L(y_i, \hat{y}_i) = L(y_i, f(x_i, \boldsymbol{W}))$$

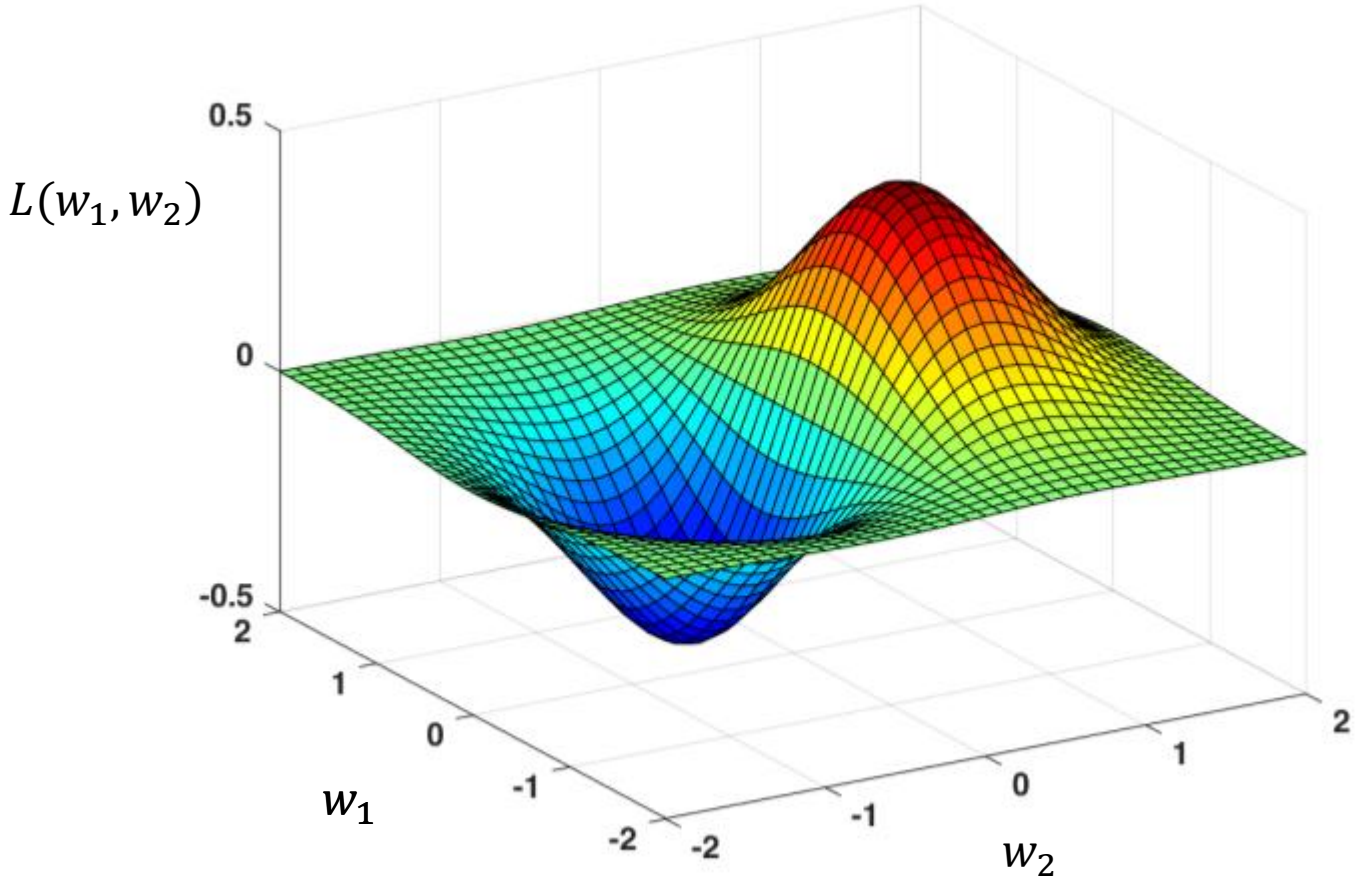
- Dado que $x_i$ são os vetores de *features* das amostras e $y_i$ são os seus valores (ou classificações) verdadeiros, os termos variáveis da função de perda são os pesos
- Os pesos ótimos, $\boldsymbol{w}^*$, correspondem aos pesos que minimizam a função de perda:

$$\boldsymbol{W}^* = \underbrace{argmin}_{\boldsymbol{W}} L(\boldsymbol{W})$$

- O treino da rede neuronal corresponde à determinação destes pesos, utilizando um algoritmo iterativo que visa encontrar esse mínimo
  - Normalmente esse algoritmo é uma variante do ***gradient descent*** (“descida do gradiente”)

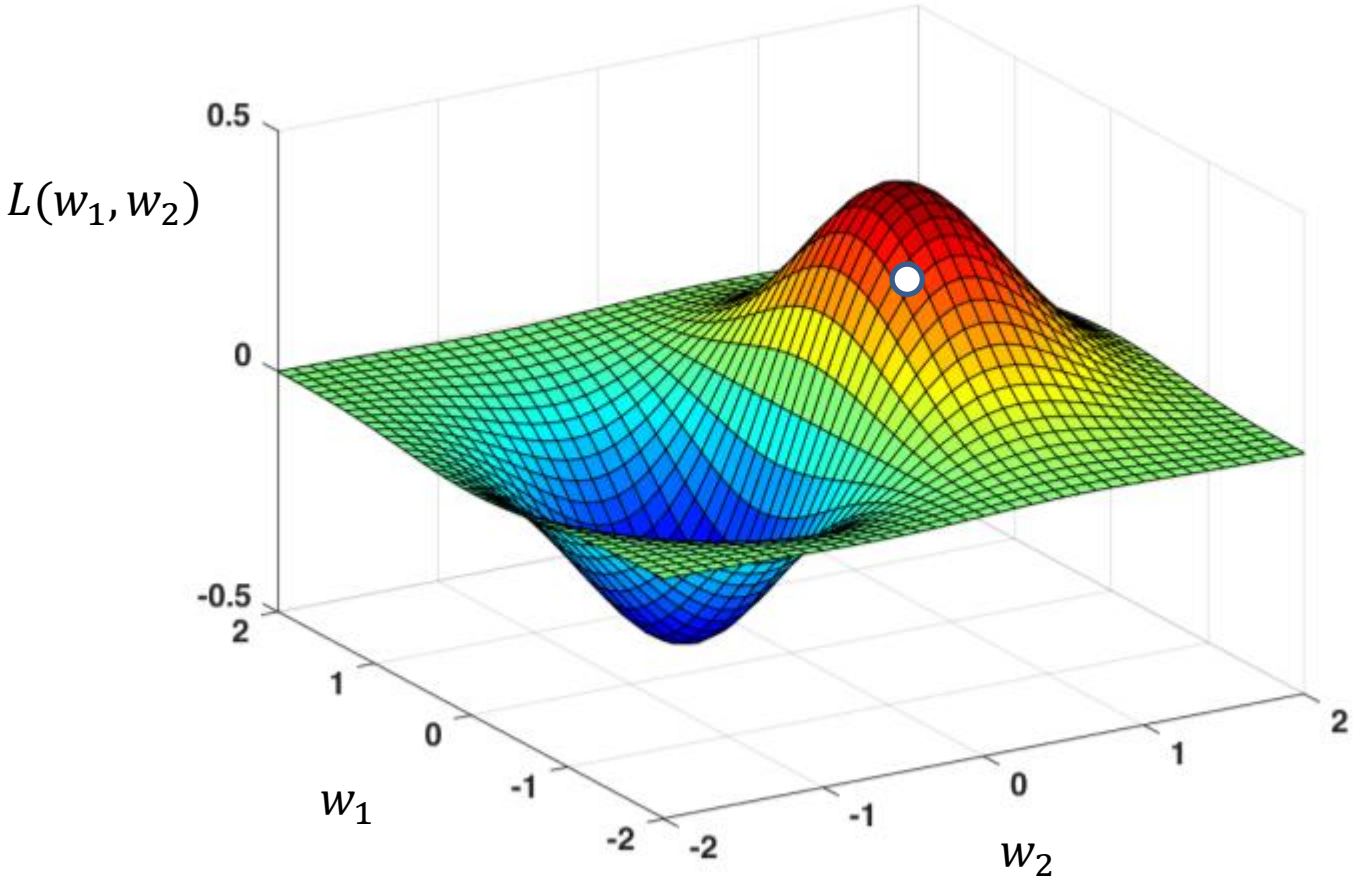
# *Gradient Descent* – Exemplo

Suponhamos que se pretendia calcular os valores dos pesos que minimizam a seguinte função de perda



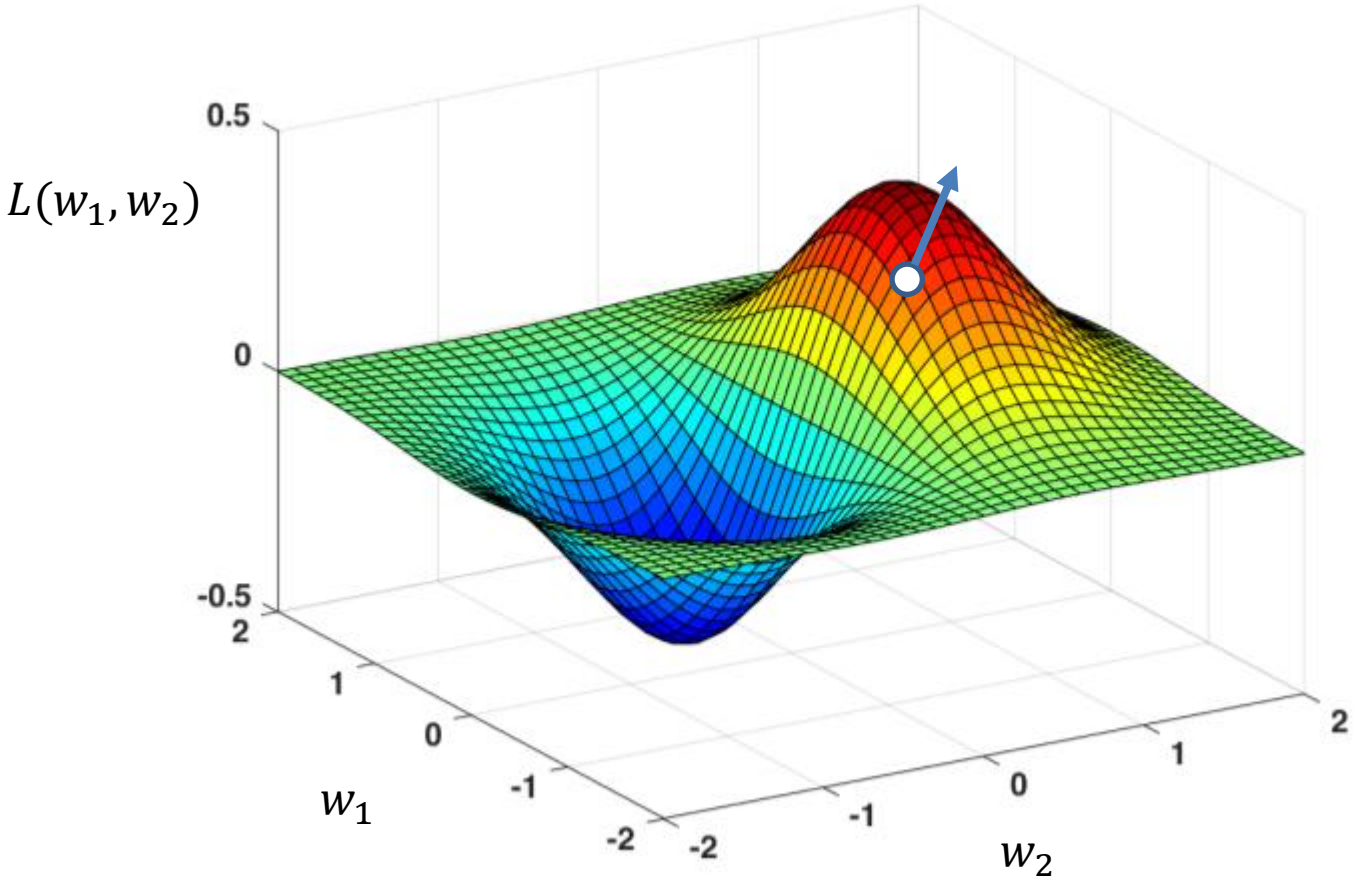
# *Gradient Descent* – Exemplo

Inicialmente começa-se por uma combinação aleatória dos valores dos pesos



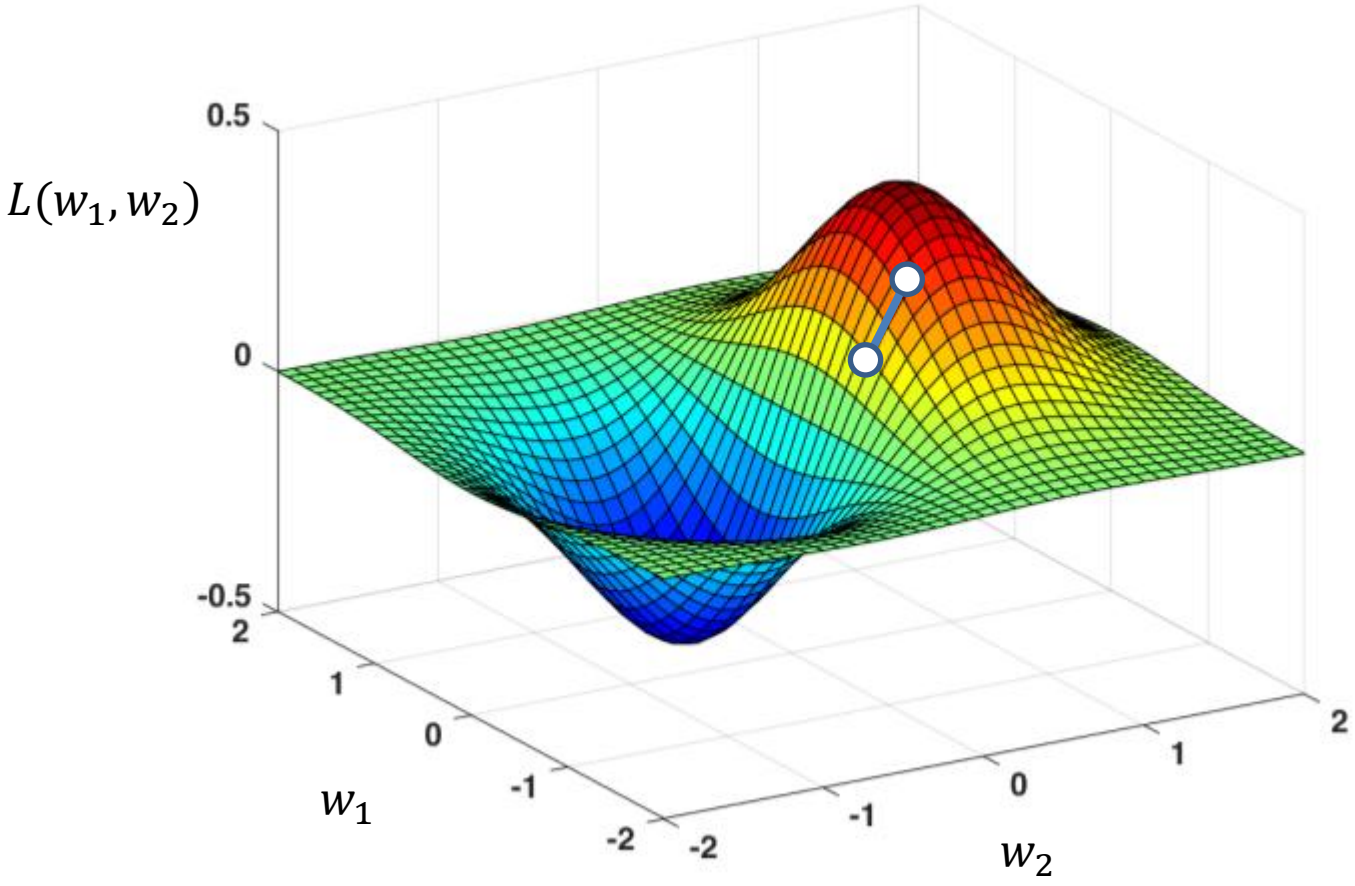
# *Gradient Descent* – Exemplo

Calcula-se o gradiente nesse ponto…



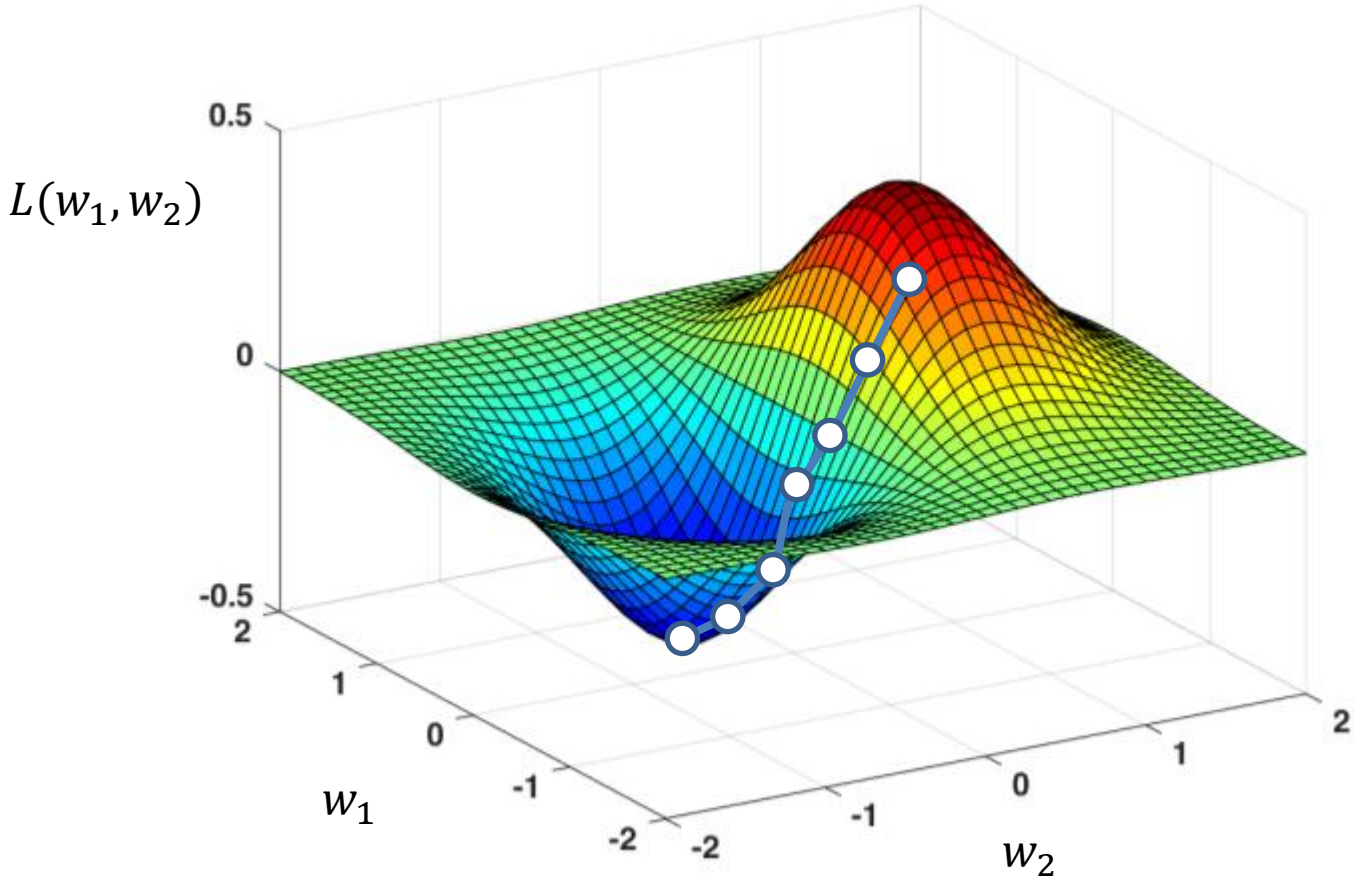
# *Gradient Descent* – Exemplo

… e caminha-se na direção contrária, obtendo-se o próximo ponto



# *Gradient Descent* – Exemplo

Vai-se sucessivamente repetindo o processo até ser atingido o valor mínimo



# *Gradient Descent* – Algoritmo

1. Inicializar aleatoriamente os valores dos pesos
2. Repetir iterativamente:

   Calcular o gradiente da função de perda, em ordem ao pesos: $\frac{\partial L(\boldsymbol{W})}{\partial \boldsymbol{W}}$

   Obter os pesos para a próxima iteração: $W^{(i+1)} = W^{(i)} - \lambda \frac{\partial L(\boldsymbol{W})}{\partial \boldsymbol{W}}$

O parâmetro $\lambda$ designa-se por **taxa de aprendizagem** (*learning rate*)

- Valores mais baixos – descida do gradiente mais lenta, o que implica um treino mais demorado; há também maior risco de ficar preso em mínimos locais
- Valores mais altos – a descida do gradiente é potencialmente mais rápida, mas corre-se o risco saltar por cima do mínimo dificultando a convergência (*overshooting*)

# Medidas de desempenho

- Taxa de acertos (*accuracy*) – habitualmente usada em problemas de classificação

$$Accuracy = \frac{Nº\ de\ previsões\ corretas}{Nº\ total\ de\ previsões\ feitas}$$

- Métricas relativas a uma determinada classe – essa classe considera-se a "positiva", e todas as outras pertencem à classe "negativa"
  - Precisão (*precision*)

$$Precision = \frac{TP}{TP + FP}$$

  - Revocação ou sensibilidade (*recall*)

$$Recall = \frac{TP}{TP + FN}$$

*TP* (*True positives*) Nº de previsões corretas para a classe positiva

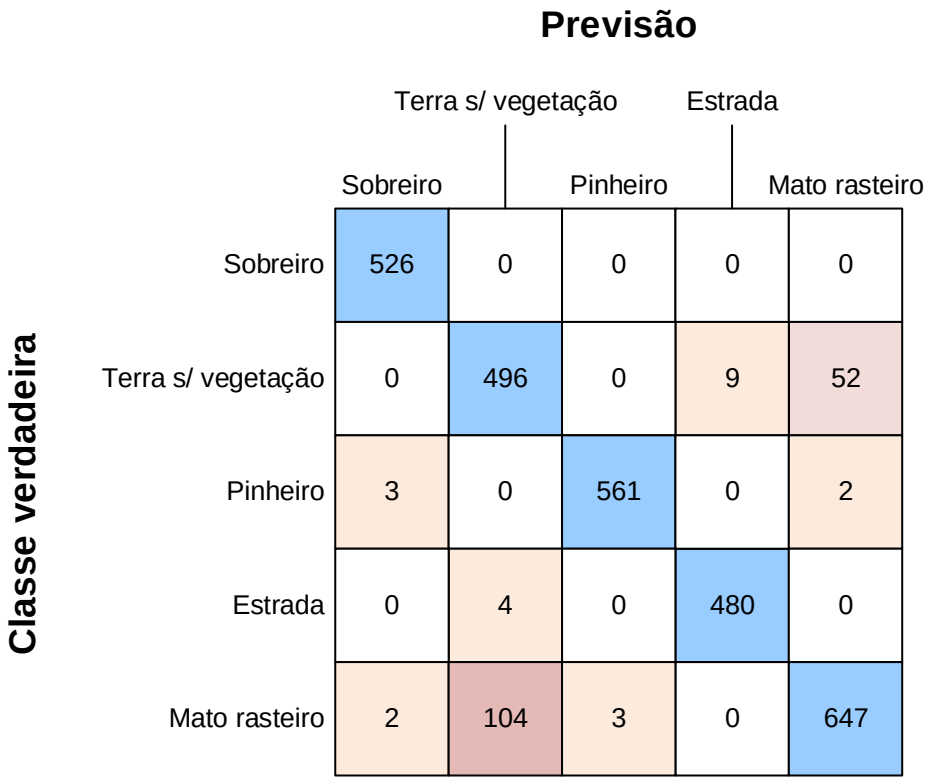
*TN* (*True negatives*) Nº de previsões corretas para a classe negativa

*FP* (*False positives*) Nº de previsões erradas para a classe negativa

*FN* (*False negatives*) Nº de previsões erradas para a classe positiva

# Medidas de desempenho

- Classificação Multiclasse – Matriz de confusão




A matriz de confusão permite analisar com maior detalhe as predições obtidas por um classificador.

Mostra as classes verdadeiras (*ground truth*) vs. predições feita pelo classificador.

É possível identificar facilmente quais as classes que o classificador tem maior dificuldade em distinguir.

Numa situação ideal, a matriz de confusão apenas teria elementos não nulos na diagonal principal. Isso significaria que todas as amostras do conjunto em análise seriam bem classificadas

# Implementação no Tensorflow (I)

- Definição do modelo da rede neuronal

Embora existam outras forma de definir o modelo da rede neuronal, a mais simples é provavelmente utilizar `Sequential(…)`, passando no argumento uma lista com as camadas que se pretendem incluir na rede neuronal a implementar

- Exemplo:

```
myModel = keras.Sequential([
    layers.Input((13,)),                    # camada de entrada – com vetores com 13 valores de features
    layers.Dense(26, activation='sigmoid'), # camada escondida – com 26 neuronios e ativação sigmoide
    layers.Dense(3, activation='softmax')   # camada de output – com 3 neurónios (para 3 classes) e ativação softmax
])
```

# Implementação no Tensorflow (II)

- Compilação e treino do modelo

```
# compilar o modelo, definindo a função de custo (loss) e o algoritmo de optimização
myModel.compile(loss = 'categorical_crossentropy',                    # função de perda
                optimizer=tf.optimizers.Adam(learning_rate=0.001),  # algoritmo de optimização – Adam é uma variante do GD
                metrics=['accuracy'])                                  # métricas (para além da loss) mostradas durante o treino


# treino do modelo – assume-se que já existe um conjunto de treino com amostras x_train e outputs y_train (targets);
# devolve um histórico com os resultados que são obtidos em cada época do treino
history = myModel.fit(x_train, y_train,
                      batch_size=32,      # nº de amostras por batch; cada batch origina uma iteração durante a otimização;
                      epochs=500,         # nº de épocas, i.e. nº de vezes que itera sobre todas as amostras de treino
                      validation_data=(x_val, y_val))    # conjunto de validação
```

# Implementação no Tensorflow (III)

- Preparação dos dados
  - Conjuntos de treino, validação e teste
    - O conjunto de dados deve sempre ser dividido pelo menos em treino e validação
    - Havendo amostras disponíveis, especifica-se também um conjunto de teste
  - Organização dos dados, admitindo $N$ amostras com $F$ atributos cada uma
    - As *features*, `x_train`, `x_val`, etc., serão matrizes de $N$ x $F$ ($N$ linhas por $F$ colunas)
    - No caso de regressão ou de classificação binária, os *targets* `y_train`, `y_val`, etc. são vetores compostos por $N$ elementos (um valor real ou um valor binário por cada amostra disponível).
    - No caso de classificação multiclasse com $C$ classes, `y_train`, `y_val`, etc. serão matrizes $N$ x $C$, onde, em cada linha, deverá estar o valor '1' na coluna que corresponde à classe da amostra, e '0' nas restantes colunas.

# Recursos

- “Neuron” image, https://en.wikipedia.org
- Tensorflow tutorials, https://www.tensorflow.org/tutorials
- A. Soleimany and A. Amini, *Introduction to Deep Learning*, online course, MIT, 2023,
  Lecture 1: “Intro to deep learning”:
  http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L1.pdf
- R. Socher, C. Manning, *Natural Language Processing with Deep Learning*, Stanford University, 2017,
  Lecture 5: Backpropagation and Project Advice
  https://www.youtube.com/watch?v=isPiE-DBagM