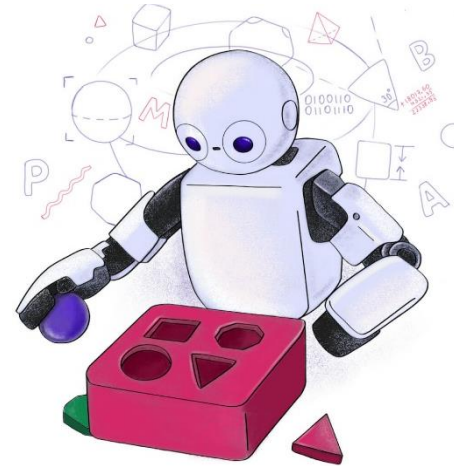


TP557 - Tópicos avançados em IoT e Machine Learning: *Classificação com DNNs*



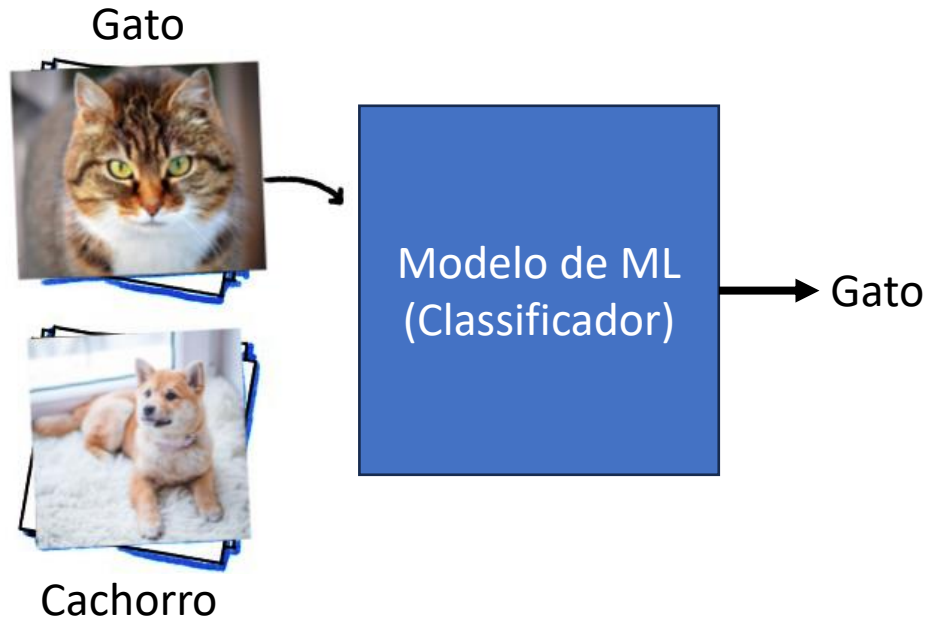
Inatel

Felipe Augusto Pereira de Figueiredo
felipe.figueiredo@inatel.br

O que vamos ver?

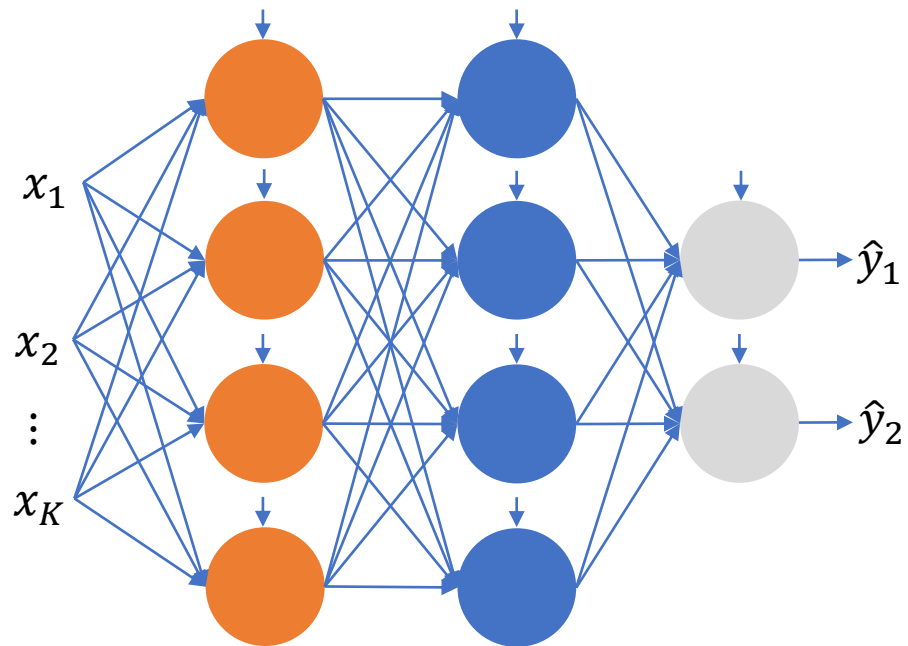
- Anteriormente, vimos como os computadores aprendem através do ajuste dos parâmetros (i.e., dos pesos) de um modelo de ML a relação entre as entradas, x , e a saída esperada, y .
- Ou seja eles *encontram um mapeamento (uma função) entre x e y* .
- Quando os valores de x *são mapeados em valores contínuos*, $y \in \mathbb{R}$, chamados o *problema de regressão*.
- Neste tópico, veremos um outro problema de mapeamento, mas desta vez, os valores de x *são mapeados em valores de um conjunto finito e discreto de valores*, $y = \{0, 1, \dots, Q - 1\}$, $y \subset \mathbb{N}$, onde Q define o número de valores de saída.
- Esse problema é chamado de *classificação*.

Classificação



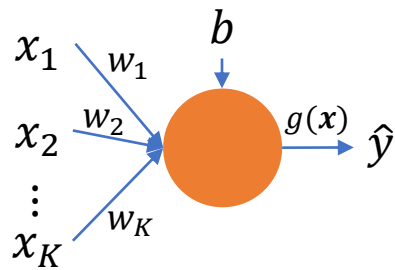
- A ideia da classificação, como o próprio nome já diz, é atribuir classes (ou grupos) aos dados de entrada.
- Por exemplo, informar o animal presente em uma imagem (i.e., conjunto de *pixels*).
- Existem vários algoritmos de ML para solucionar este problema:
 - Regressão Logística e *Softmax*;
 - Árvores de Decisão;
 - Máquinas de vetores de suporte;
 - etc.
- Porém, neste curso, focaremos no uso de **redes neurais artificiais** para classificação.

Classificação com redes neurais



- Na sequência, veremos como ***redes neurais com múltiplas camadas podem também resolver problemas de classificação***, bastando fazer algumas alterações na arquitetura da rede.
- Basicamente, precisamos alterar a ***função de ativação da camada de saída*** e a ***função de erro/loss***.

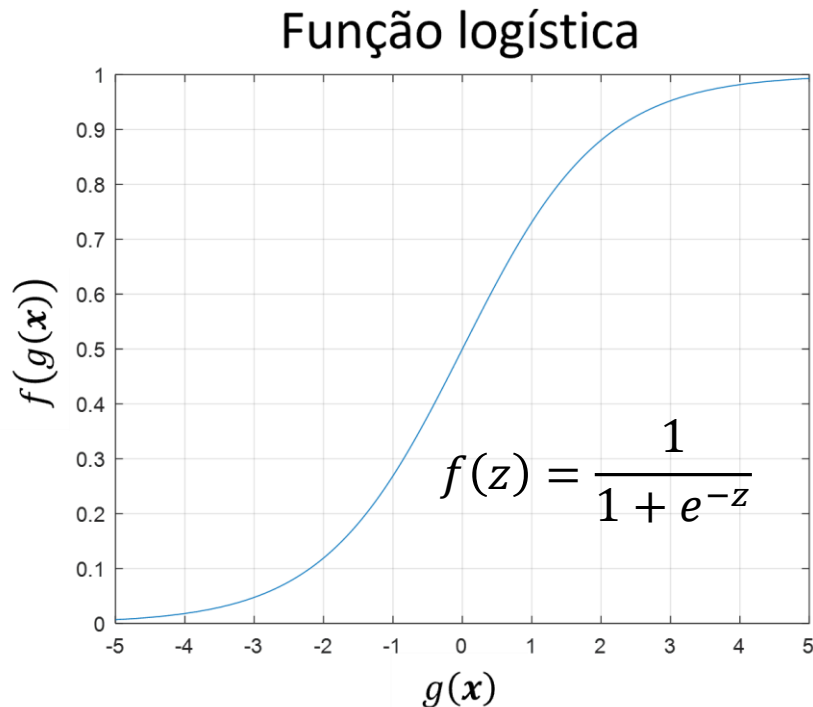
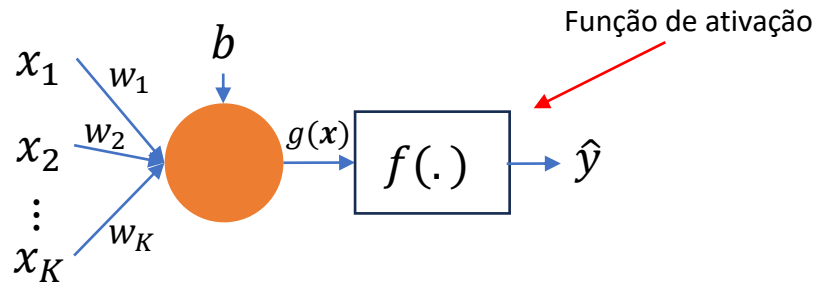
Classificação com neurônios



$$\hat{y} = g(\mathbf{x}) = \left(\sum_{k=1}^K w_k x_k \right) + b$$

- Até agora, usamos um neurônio (**sem função de ativação**, $f(\cdot)$) para mapear valores de \mathbf{x} em y , ou seja, encontrar uma **relação linear** entre eles.
 - Lembrando que, em geral, \mathbf{x} é um vetor contendo vários atributos.
- Entretanto, os neurônios podem ser usados para **classificação binária** se usarmos uma função de ativação, $f(\cdot)$, do tipo sigmoide (ou logística), por exemplo.

Classificação com neurônios

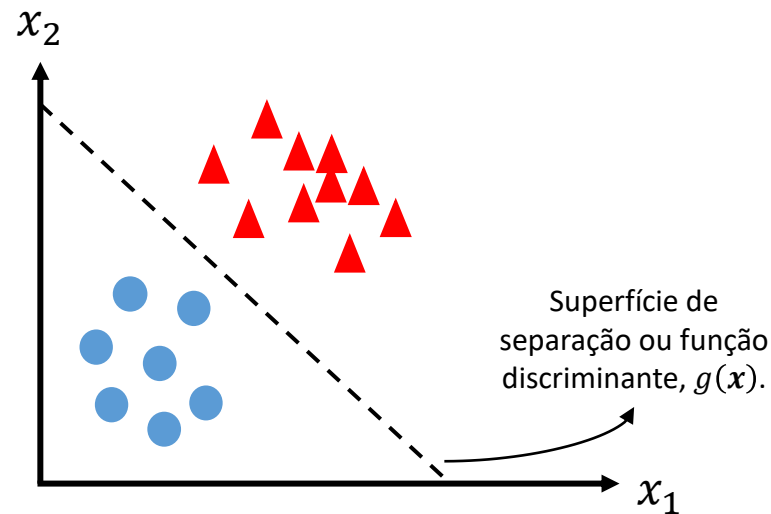


- Ao usarmos uma função de ativação, o modelo matemático do neurônio passa a ser expresso por

$$\hat{y} = f(g(\mathbf{x})) = f\left(\left(\sum_{k=1}^K w_k x_k\right) + b\right).$$

- A função de ativação logística, $f(\cdot)$, é uma **suavização da função degrau** e, portanto, **derivável**, que mapeia os valores de $g(\mathbf{x})$ (intervalo infinito) em valores no intervalo $[0, 1]$.
- Notem que $g(\mathbf{x})$ continua sendo a equação de um hiperplano.

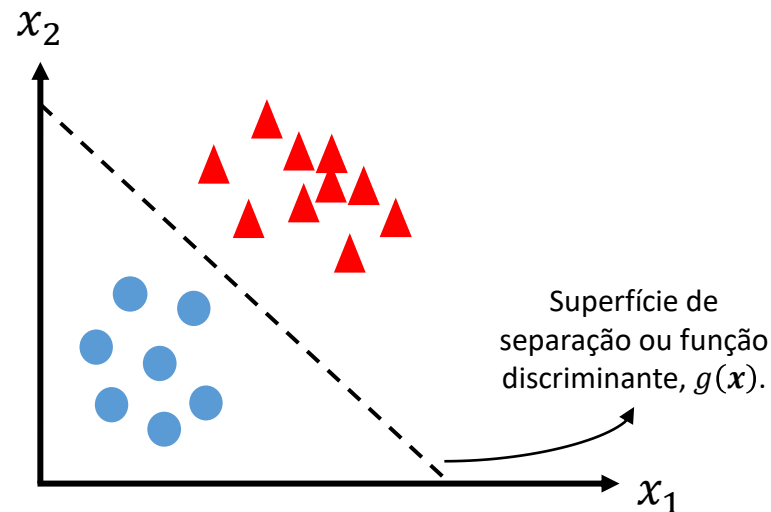
Classificação com neurônios



- ▲ Classe 0 (Classe negativa)
- Classe 1 (Classe positiva)

- Quando o neurônio opera como um **classificador binário**, a combinação linear definida por $g(x)$, define uma **superfície de separação linear** entre **dois grupos** distintos de dados (i.e., classes), **ao invés de uma função que aproxima o comportamento dos dados**.

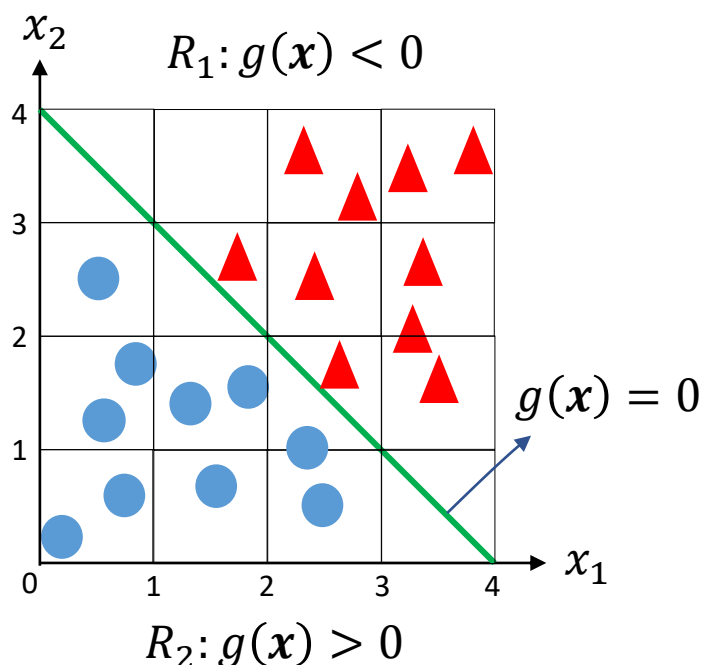
Classificação com neurônios



- ▲ Classe 0 (Classe negativa)
- Classe 1 (Classe positiva)

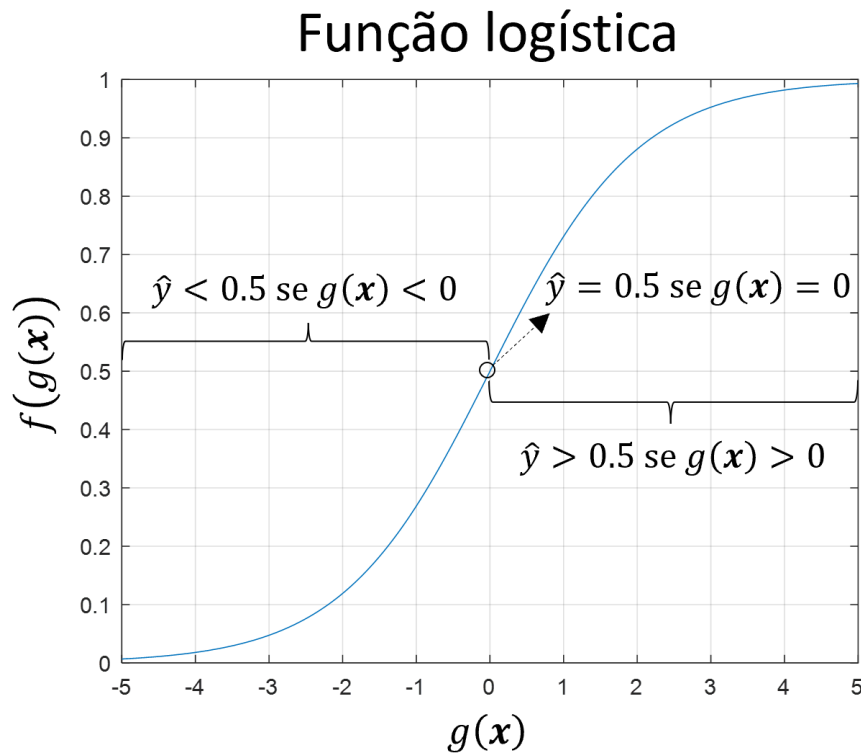
- O neurônio separa as classes usando **hiperplanos** (retas, planos, etc.).
- Nesse contexto, a função $g(x)$ é chamada também de **função discriminante**, pois seus valores de saída são usados para separar as classes.

Função discriminante, $g(\mathbf{x})$



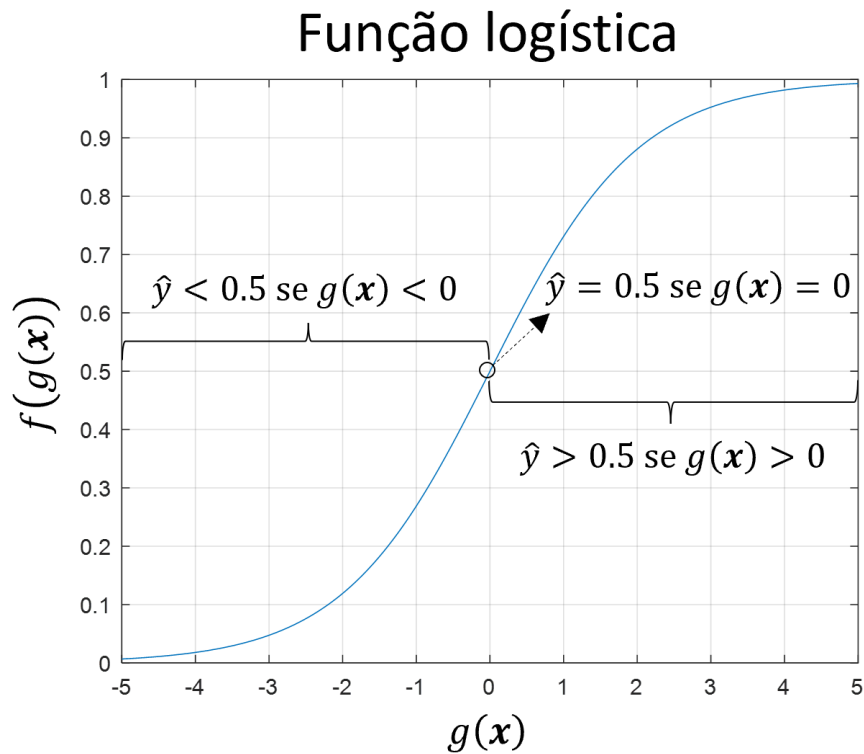
- Vamos supor a seguinte função discriminante: $g(\mathbf{x}) = 4 - x_1 - x_2$, a qual é mostrada na figura ao lado.
- Agora vamos encontrar as **regiões de decisão** substituindo alguns valores nela.
 - $x_1 = 1$ e $x_2 = 1$ resulta em $g(\mathbf{x}) > 0$.
 - ✓ Região da classe **positiva**, C_2 .
 - $x_1 = 3$ e $x_2 = 3$ resulta em $g(\mathbf{x}) < 0$.
 - ✓ Região da classe **negativa**, C_1 .
 - $x_1 = 1$ e $x_2 = 3$ resulta em $g(\mathbf{x}) = 0$.
 - ✓ **Indeterminação**: não podemos afirmar a qual classe o exemplo pertence.
 - ✓ Podemos atribuir arbitrariamente a uma das duas classes ou escolher a classe que possui maior número de exemplos.

Classificação com neurônios



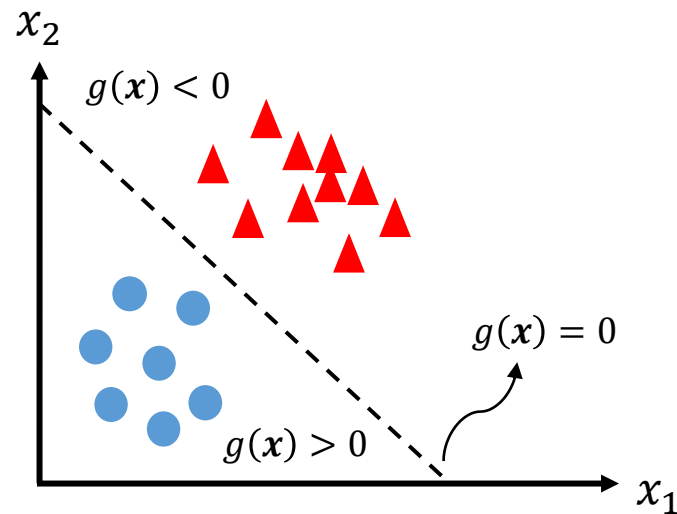
- Ao passar $g(x)$ pela função de ativação, $f(\cdot)$, temos os seguintes valores de saída:
$$f(g(x)) = \begin{cases} 0 \leq \hat{y} < 0.5, & \text{se } g(x) < 0 \\ 0.5 < \hat{y} \leq 1, & \text{se } g(x) > 0 \\ 0.5, & \text{se } g(x) = 0 \end{cases}$$
- Esses valores de saída do neurônio são **interpretados como sendo a probabilidade** (pois valores estão no intervalo $[0, 1]$) de x pertencer à **classe positiva**.
- Quanto **mais distante** de $g(x)$ estiver x , **maior será a probabilidade dele pertencer a uma das duas classes**.

Classificação com neurônios



- A probabilidade da classe negativa é obtida através do complemento de \hat{y} , i.e., $1 - \hat{y}$.
- Notem \hat{y} **pode assumir qualquer valor no intervalo** $[0, 1]$, ou seja, esse é um **problema de regressão e não de classificação!**
- **Então como obtemos um classificador?**

Classificação com neurônios

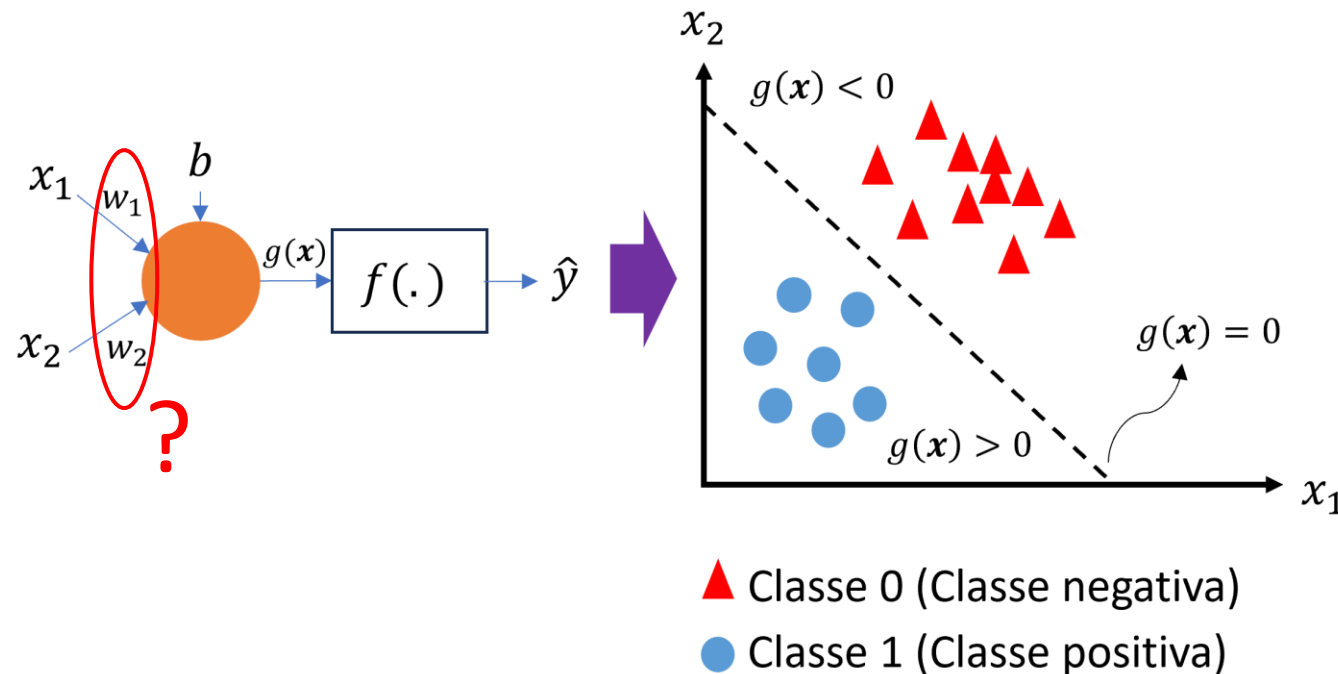


- ▲ Classe 0 (Classe negativa)
- Classe 1 (Classe positiva)

- Para classificar, *quantizamos os valores de saída da função de ativação*:
$$C = \begin{cases} 0, & \hat{y} < 0.5, \\ 1, & \hat{y} > 0.5, \\ \text{uma ou outra,} & \hat{y} = 0.5. \end{cases}$$
- Em geral, quando $\hat{y} = 0.5$, atribui-se x à *classe mais frequente* ou a *uma das duas classes de forma arbitrária*.

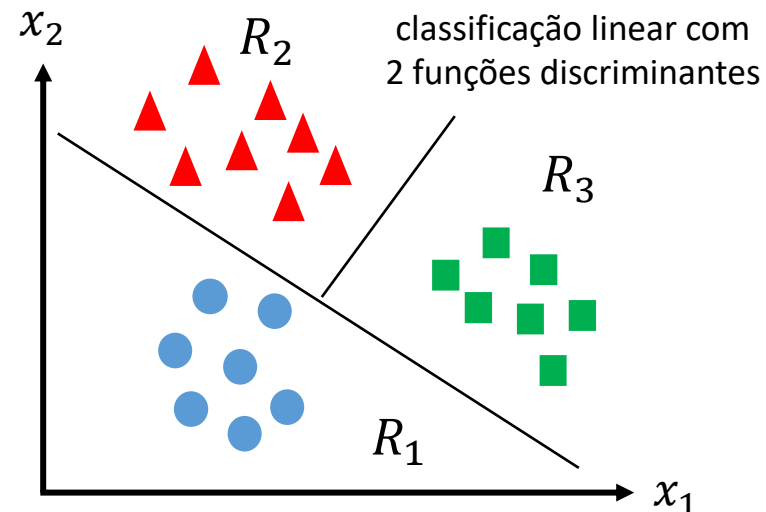
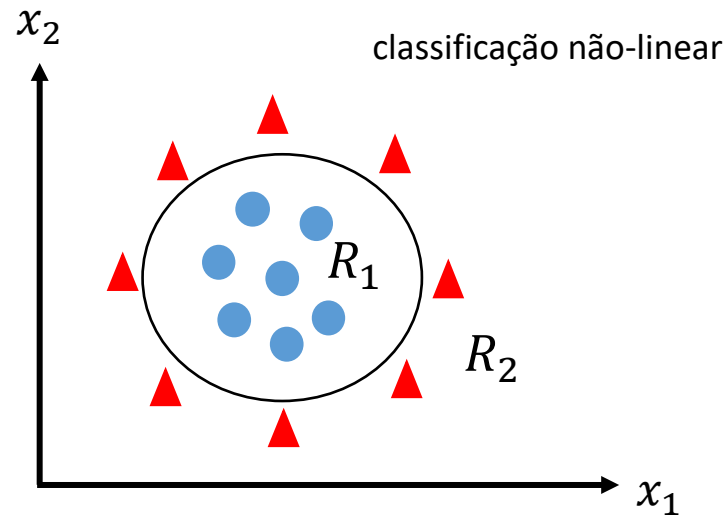
Classificação com neurônios

- O **objetivo** do treinamento do classificador é **encontrar os pesos** de $g(\mathbf{x})$ de forma que as **classes sejam separadas da melhor forma possível**.
- Ou seja, assim como na regressão, o objetivo é **minimizar uma função de erro**.



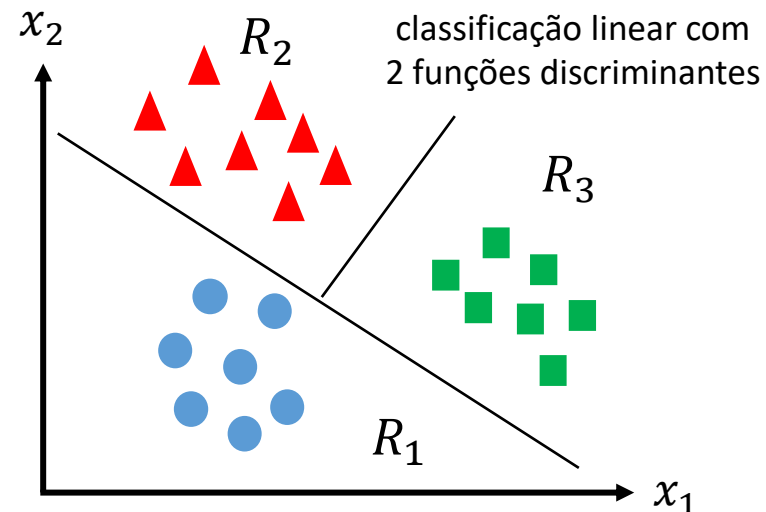
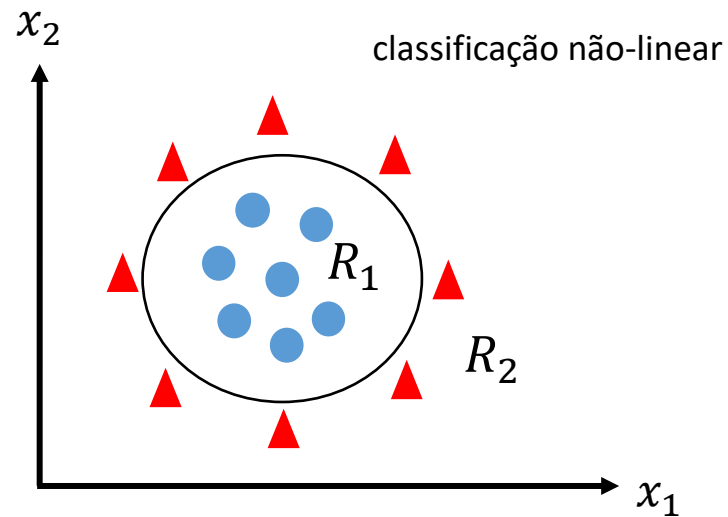
***Mas e se tivermos mais de duas classes ou
elas não puderem ser separadas por
hiperplanos?***

Classificação com redes neurais



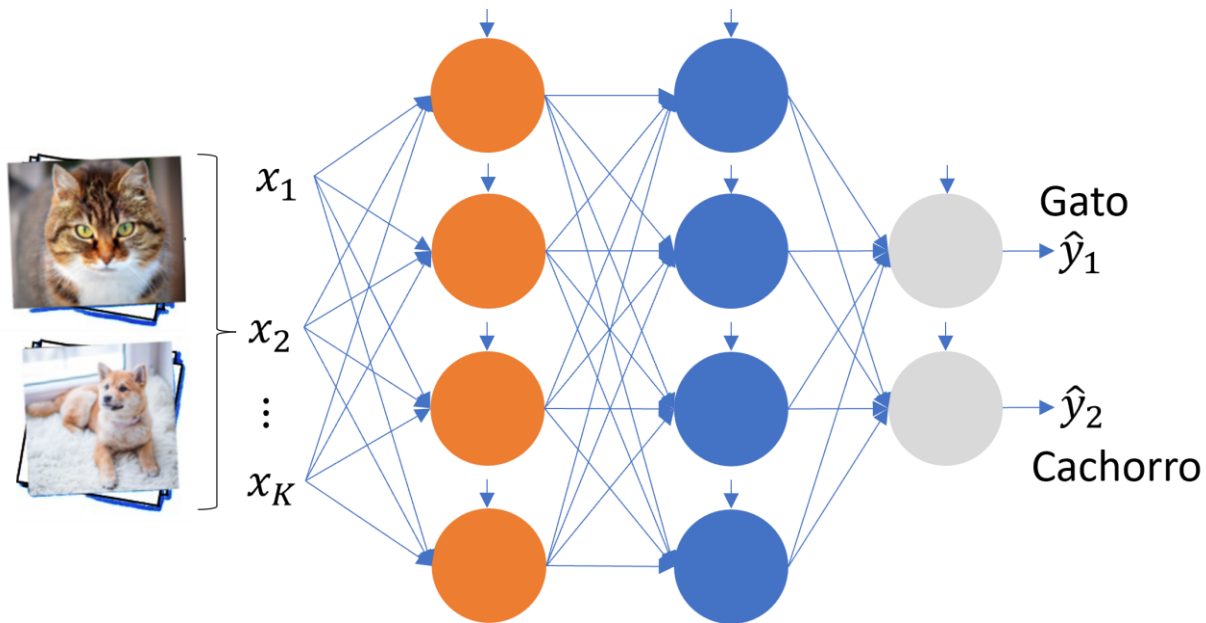
- Se precisarmos resolver problemas com ***mais classes*** ou ***mais complexos*** (e.g., que ***exijam superfícies de separação não-lineares ou várias delas***), podemos ***agrupar vários neurônios e usar uma rede neural***.
- Usamos uma ou mais camadas escondidas para encontrar uma superfície de separação adequada.

Classificação com redes neurais



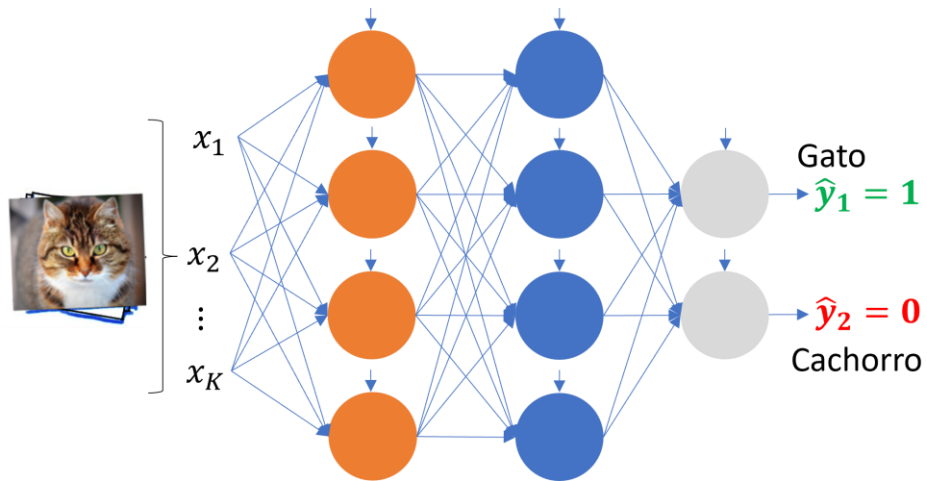
- No caso com múltiplas classes, teremos **vários neurônios na camada de saída** da rede, **um para cada classe** do problema de classificação.
- **Cada saída** da rede neural dá a **probabilidade de uma das classes**.
- **OBS.:** Mesmo no caso com duas classes, podemos criar uma rede com duas saídas, onde cada uma delas dá a probabilidade de uma das classes.

Classificação com redes neurais

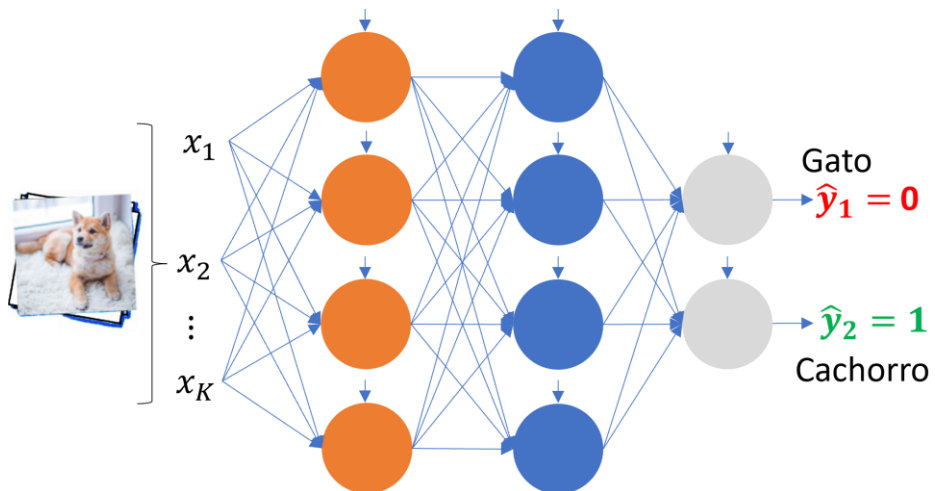


- Portanto, se quisermos projetar uma rede neural capaz de distinguir entre gatos e cachorros, poderíamos ter ***um neurônio de saída representando gatos e outro representando cães.***
- ***Cada neurônio*** de saída irá apresentar a ***probabilidade de uma imagem de entrada pertencer a uma das classes.***

Classificação com redes neurais



- Então, se uma *imagem de um gato* for *passada para a rede*, o *neurônio que representa os gatos* terá um *alto valor de probabilidade* (teoricamente 1) e o que representa os *cachorros* terá um *baixo valor de probabilidade* (teoricamente 0).



- O contrário ocorreria se imagem de um cachorro fosse passada para a rede.

Classificação com redes neurais

Dados de
entrada



Rótulo

[1, 0]



[0, 1]

- As *saídas (teóricas) da rede* nos fornecem uma maneira de *rotular* cães e gatos fazendo com que o *rótulo* de uma imagem seja um *vetor contendo a saída desejada de cada um dos neurônios da camada de saída*.

Classificação com redes neurais

Dados de
entrada



Rótulo

[1, 0]



[0, 1]

- Portanto, se o primeiro neurônio da camada de saída representa gatos e o segundo cães, podemos dizer que o **rótulo** (i.e., saída esperada) **para imagens de gatos é dado pelo vetor: [1, 0]**.
- E o rótulo para imagens de cães é dado pelo vetor [0, 1].
- Essa forma de representar os rótulos é chamada de **codificação one-hot**.
 - Vetor onde um único elemento é igual a 1 (indicando a classe verdadeira) e todos os outros são iguais a 0.

Classificação com redes neurais

- E se quiséssemos usar uma rede neural para reconhecer dígitos escritos à mão?
 - Esse é um problema com 10 classes, os dígitos de 0 a 9.
- Quantos neurônios seriam necessários na camada de saída?
- Como seria a codificação ***one-hot*** dos rótulos nesse caso?

Classificação com redes neurais

0 → [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
1 → [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
2 → [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
3 → [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
4 → [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
5 → [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
6 → [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
7 → [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
8 → [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
9 → [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

- Precisaríamos de **10 neurônios**, 1 para cada um dos 10 possíveis dígitos (0 a 9).
- O **rótulo** seria um **vetor com 10 elementos**.
- Porém, **apenas um dos elementos seria diferente de 0**, aquele que identifica a qual dígito x pertence.
- Vejamos na sequência um código que usa uma rede neural para reconhecer dígitos escritos à mão.

A base de dados

```
import tensorflow as tf
```

```
data = tf.keras.datasets.mnist  
(X_train, y_train), (X_test, y_test) = data.load_data()
```

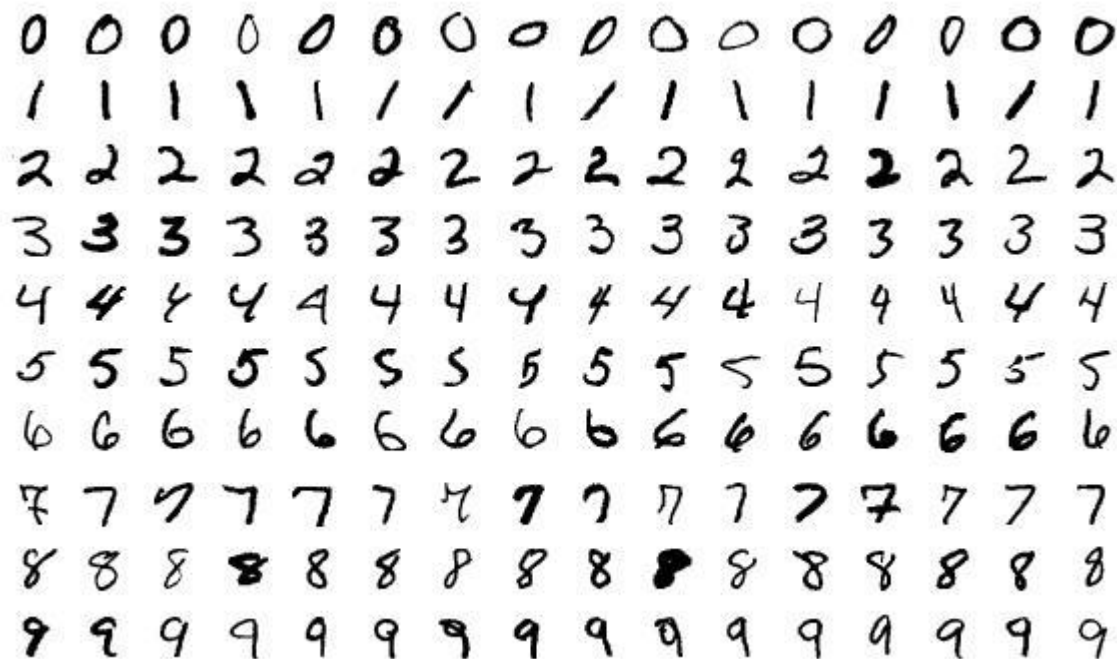
```
X_train = X_train / 255.0
```

```
X_test = X_test / 255.0
```

```
model = tf.keras.models.Sequential(  
    [tf.keras.layers.Flatten(input_shape=(28, 28)),  
     tf.keras.layers.Dense(20, activation=tf.nn.relu),  
     tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

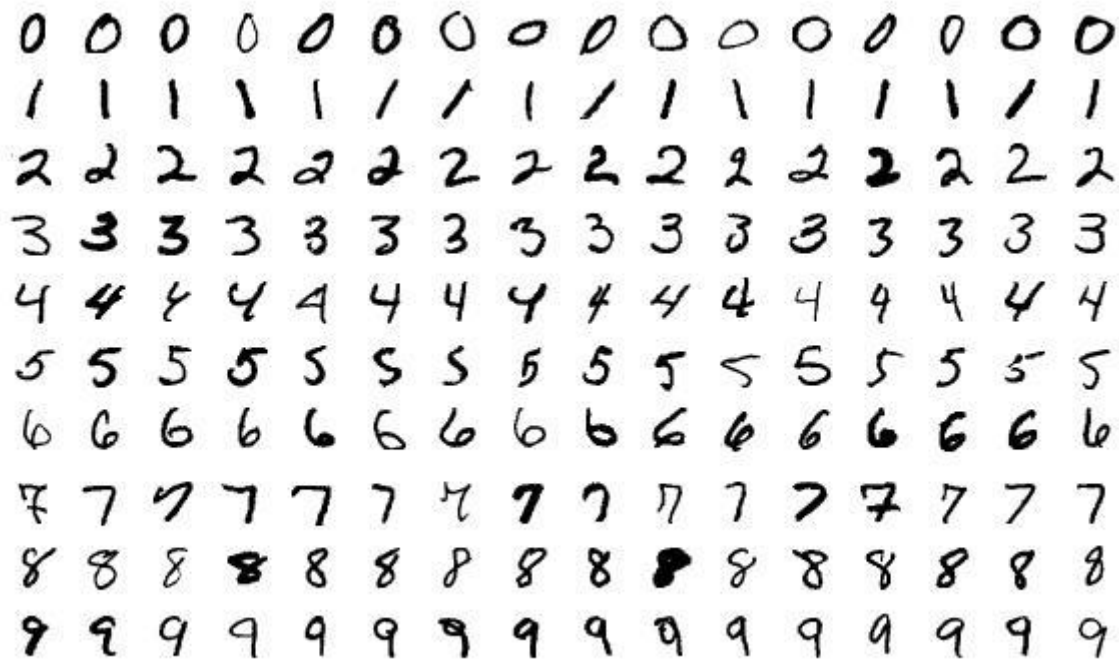
- Usaremos a base de dados com imagens de dígitos escritos à mão do MNIST (*Modified National Institute of Standards and Technology*).
- O TensorFlow disponibiliza uma função (i.e., *load_data()*) que baixa e divide os dados em conjuntos de treinamento e teste.

A base de dados



- A base de dados possui um conjunto de ***treinamento de 60.000 imagens*** e um ***conjunto de teste de 10.000 imagens***.
- São ***imagens em escala de cinza*** com 28x28 pixels dos dígitos de 0 a 9.
 - Notem que as imagens são *arrays bidimensionais*.

A base de dados



- Os **pixels** são representados por **valores inteiros não sinalizados de 8 bits**, i.e., variam de 0 a 255.
- Cada **pixel será um atributo de entrada da rede neural**.
- Os rótulos são valores de 0 a 9.

Normalização dos dados

```
import tensorflow as tf
```

```
data = tf.keras.datasets.mnist  
(X_train, y_train), (X_test, y_test) = data.load_data()
```

```
X_train = X_train / 255.0  
X_test = X_test / 255.0
```

```
model = tf.keras.models.Sequential(  
    [tf.keras.layers.Flatten(input_shape=(28,28)),  
     tf.keras.layers.Dense(20, activation=tf.nn.relu),  
     tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

- Para ***melhorar a estabilidade*** do treinamento e ***aumentar a velocidade de convergência***, nós normalizamos os valores das imagens.
 - Em geral, as redes neurais se ***comportam bem melhor se os dados são escalonados***.
- Normalizamos os valores dos pixels de forma que fiquem no intervalo entre 0 e 1.
- Para fazer isso, dividimos os valores por 255.



Definindo a rede neural

```
import tensorflow as tf
```

```
data = tf.keras.datasets.mnist  
(X_train, y_train), (X_test, y_test) = data.load_data()
```

```
X_train = X_train / 255.0  
X_test = X_test / 255.0
```

```
model = tf.keras.models.Sequential(  
    [tf.keras.layers.Flatten(input_shape=(28,28)),  
     tf.keras.layers.Dense(20, activation=tf.nn.relu),  
     tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

- Como fizemos antes, definimos a rede ***empilhando*** (i.e., listando) algumas camadas em uma rede ***sequencial*** (i.e., *dense feed forward*).

Projetar
modelo

Definindo a rede neural

```
import tensorflow as tf
```

```
data = tf.keras.datasets.mnist  
(X_train, y_train), (X_test, y_test) = data.load_data()
```

```
X_train = X_train / 255.0  
X_test = X_test / 255.0
```

```
model = tf.keras.models.Sequential(  
    [tf.keras.layers.Flatten(input_shape=(28,28)),  
     tf.keras.layers.Dense(20, activation=tf.nn.relu),  
     tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

- *Não existe uma regra para definirmos a arquitetura da rede* (i.e., número de camadas, nós, otimizador, função de erro, etc.), o *mais indicado é testar diferentes arquiteturas*, de preferência com *técnicas de otimização hiperparamétrica*.
- Assim, apenas para ilustrar como realizar classificação com redes neurais, iremos adotar uma arquitetura qualquer.

Projetar
modelo

Camada de achatamento

```
import tensorflow as tf

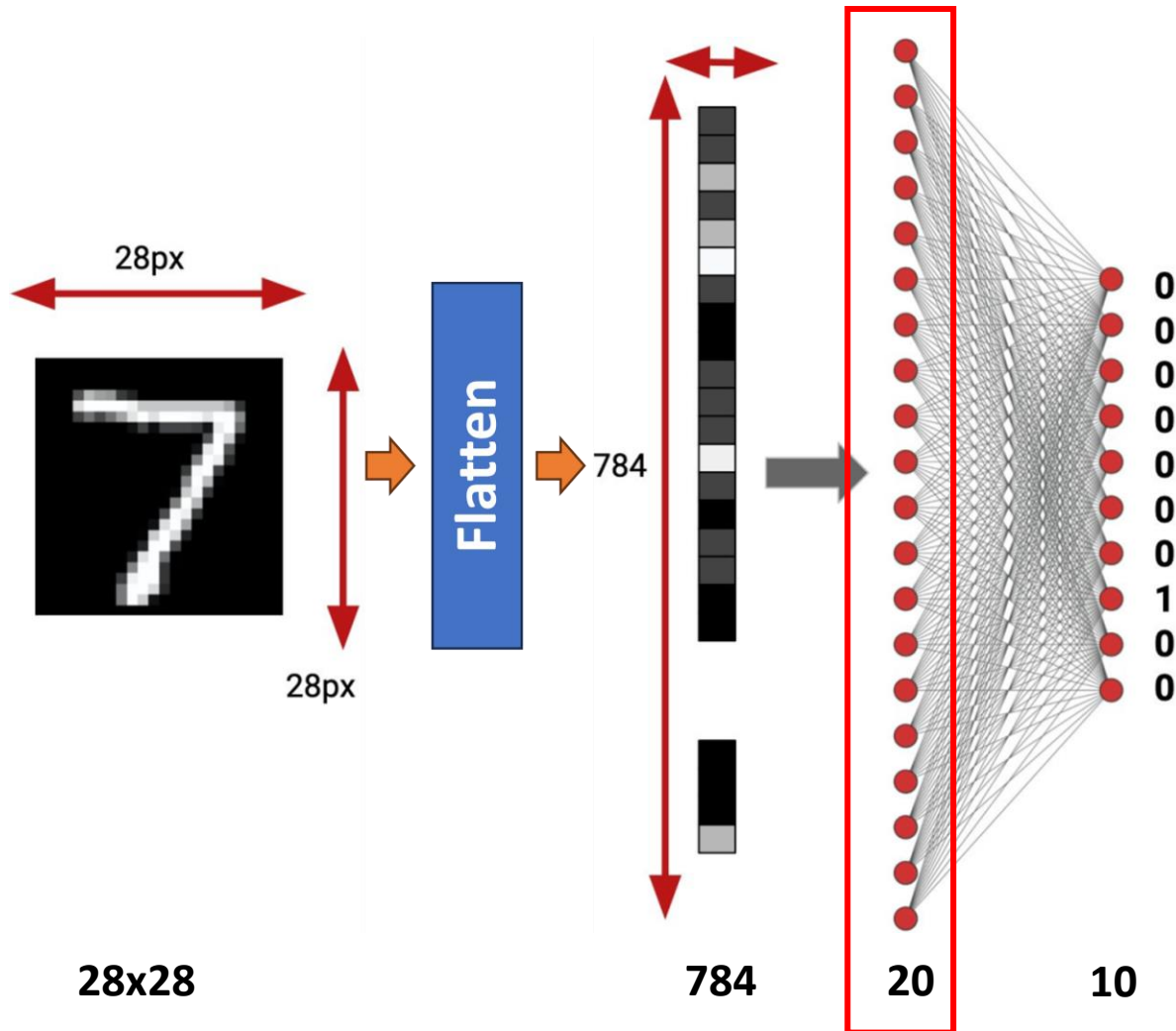
data = tf.keras.datasets.mnist
(X_train, y_train), (X_test, y_test) = data.load_data()

X_train = X_train / 255.0
X_test = X_test / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(20, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

- Como as imagens são *arrays* bidimensionais com dimensão 28x28 pixels, a primeira operação que precisamos realizar é *transformá-las em arrays unidimensionais* com 784 pixels (i.e., $28 \times 28 = 784$ pixels).
- Essa camada *não tem parâmetros a serem aprendidos durante o treinamento*, ela só *redimensiona* os dados de entrada.
- Assim, a primeira camada da rede, *Flatten*, realiza uma operação de *achatamento* das *arrays* 2D de entrada.

Por que achatamos as imagens?



- Notem que a camada (oculta) seguinte à *Flatten* tem 20 neurônios.
- Essa camada oculta espera como entrada uma ***array unidimensional***.
- Por isso precisamos que as imagens sejam transformadas em *arrays* com uma única dimensão e 784 elementos.

Camada de achatamento

```
import tensorflow as tf

data = tf.keras.datasets.mnist
(X_train, y_train), (X_test, y_test) = data.load_data()

X_train = X_train / 255.0
X_test = X_test / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(20, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

- Como a camada *Flatten* é a primeira camada, definimos nela as **dimensões de entrada**.
- Definimos que a camada deve esperar por *arrays* de entrada com dimensões 28x28.
- Se ela receber *arrays* com dimensões diferentes, um erro será lançado.
- **OBS.:** Lembrem-se que não definimos o número de exemplos de entrada pois ele varia com o tamanho do *mini-batch*.

Camada oculta

```
import tensorflow as tf

data = tf.keras.datasets.mnist
(X_train, y_train), (X_test, y_test) = data.load_data()

X_train = X_train / 255.0
X_test = X_test / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(20, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

- Na sequência, definimos a primeira e única *camada oculta densa* desta rede.
- Como discutido anteriormente, ela possui 20 neurônios.
 - Esse número foi escolhido de forma arbitrária*.

*Mais tarde vocês explorarão outras arquiteturas.

Camada oculta

```
import tensorflow as tf

data = tf.keras.datasets.mnist
(X_train, y_train), (X_test, y_test) = data.load_data()

X_train = X_train / 255.0
X_test = X_test / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(20, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

- Com *poucos nós*, a *rede pode não ter capacidade suficiente para aprender as características* contidas nas imagens.
- Por outro lado, *com muitos nós* a *rede pode se especializar demais* ou ficar *muito lenta* para aprender.
- A camada usa função de ativação ReLU, que iremos detalhar em seguida.

Camada de saída

```
import tensorflow as tf

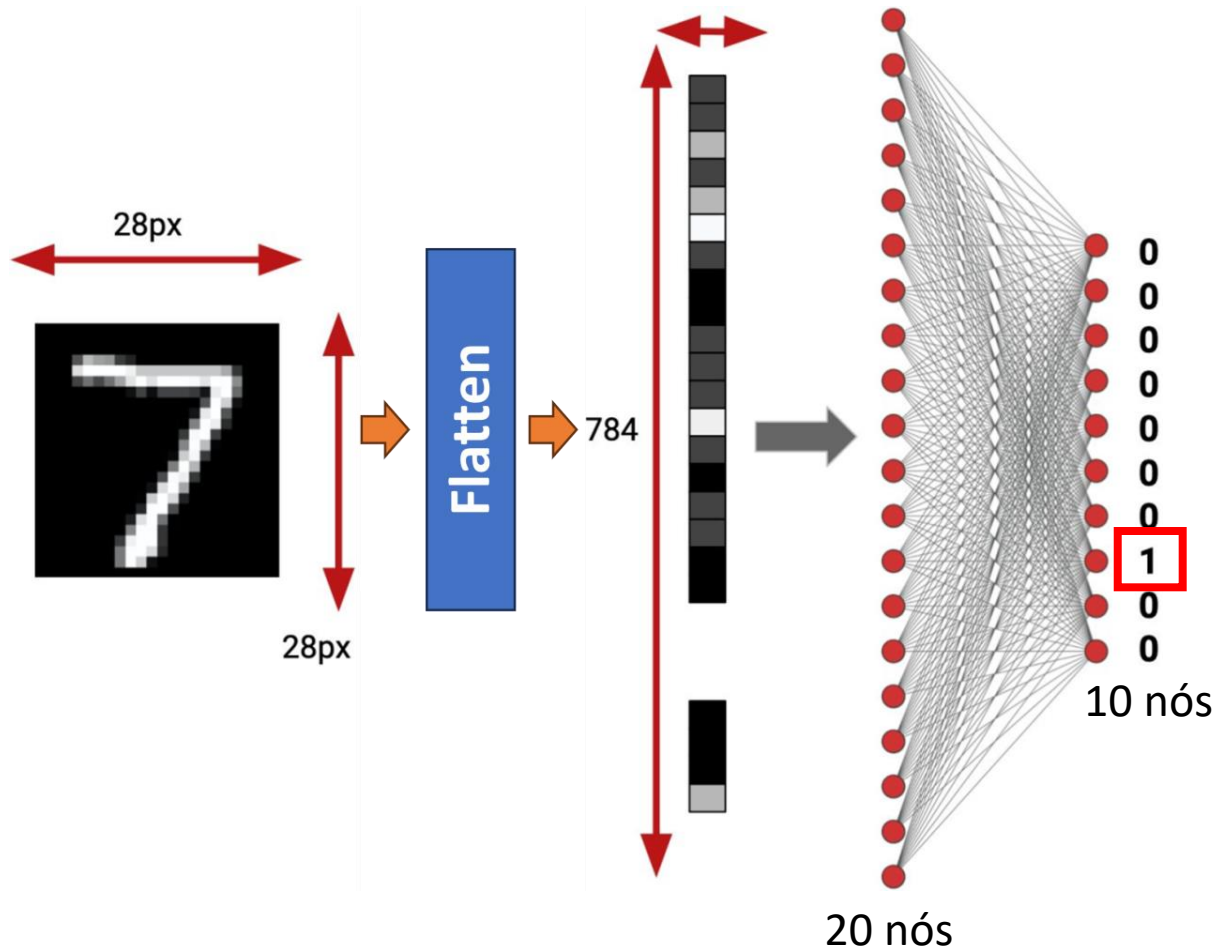
data = tf.keras.datasets.mnist
(X_train, y_train), (X_test, y_test) = data.load_data()

X_train = X_train / 255.0
X_test = X_test / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(20, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

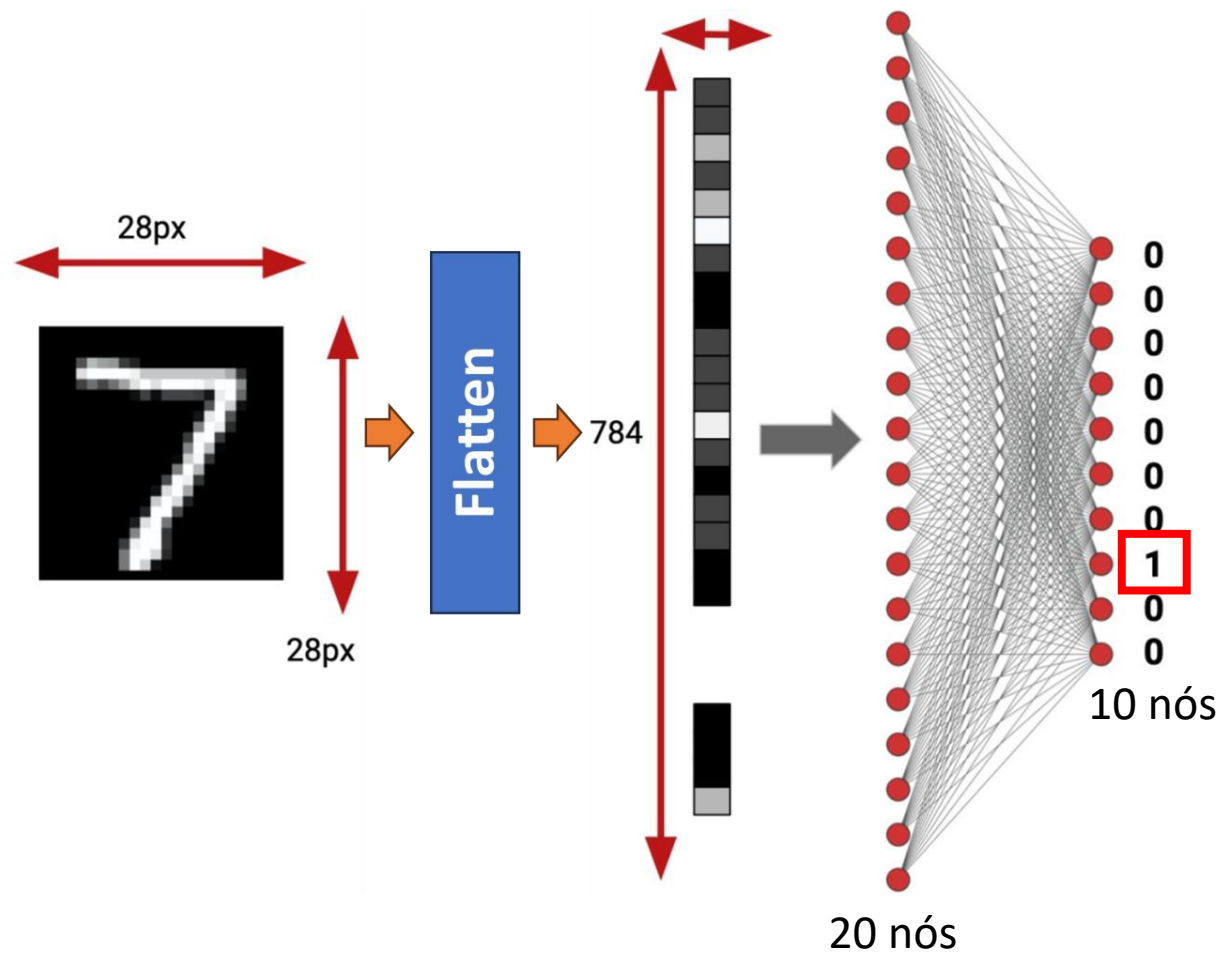
- Como temos 10 possíveis classes, a camada de saída possui 10 neurônios, um para *calcular a probabilidade de cada uma das classes* do problema.
- Ela usa função de ativação *Softmax*, a qual discutiremos em detalhes a seguir.

A rede neural



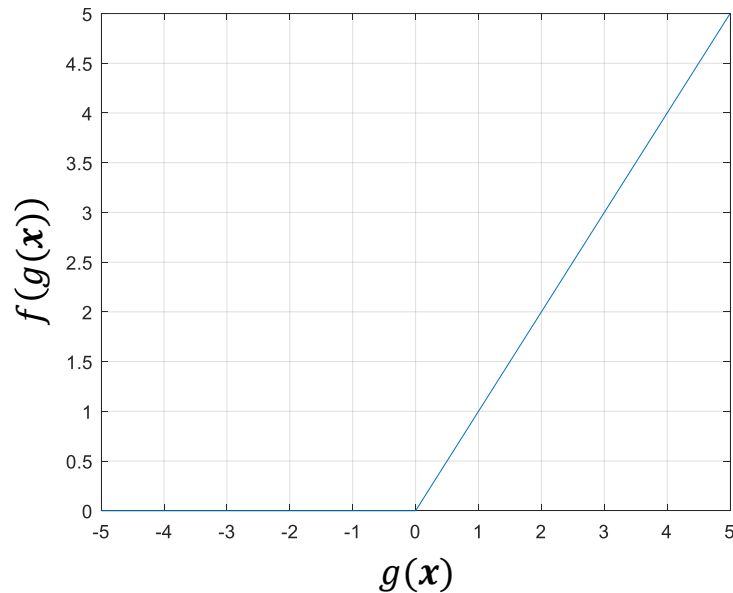
- A figura mostra a **rede neural densa** que criamos, com 20 neurônios na camada oculta e 10 na de saída.
- Cada neurônio está conectado a cada entrada ou neurônio da camada anterior.

A rede neural



- Assim, se alimentarmos a rede com os *pixels* de uma **imagem contendo um número 7**.
- Então, desejaríamos que o **neurônio de saída que representa o número 7 fosse o que tivesse o maior valor de probabilidade** (em teoria 1).
- Portanto, o **rótulo dessa imagem é um vetor com 10 elementos** contendo 0s, com exceção do elemento que representa o dígito 7, o qual terá o valor 1.

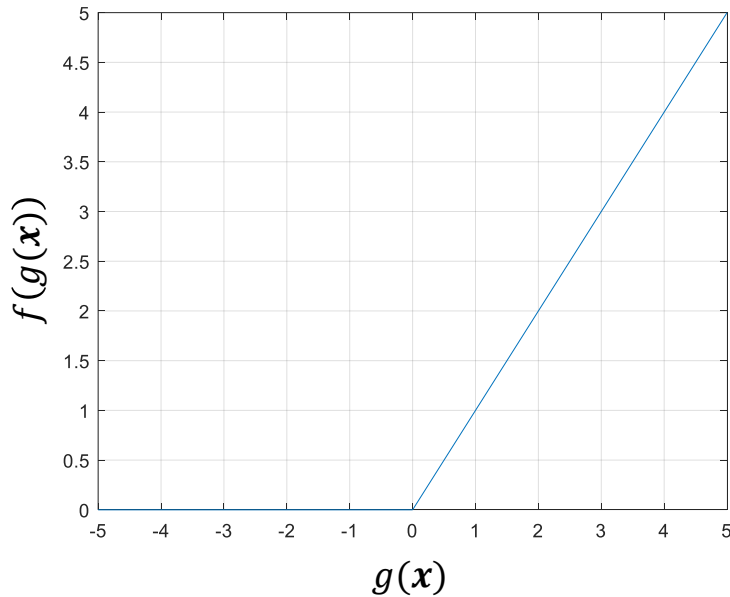
Funções de ativação: ReLU



$$\hat{y} = f(g(x)) = \max(0, g(x))$$

- Como vimos, a camada oculta usa funções de ativação ***Rectified Linear Unit*** (ReLU) em seus neurônios.
- É uma ***função não-linear*** em que sua saída é igual 0 quando $g(x) \leq 0$ e o próprio $g(x)$ quando $g(x) > 0$.
- É uma das funções mais amplamente utilizadas em redes neurais.
- Suas principais ***vantagens*** são a sua ***simplicidade e eficiência computacional***.

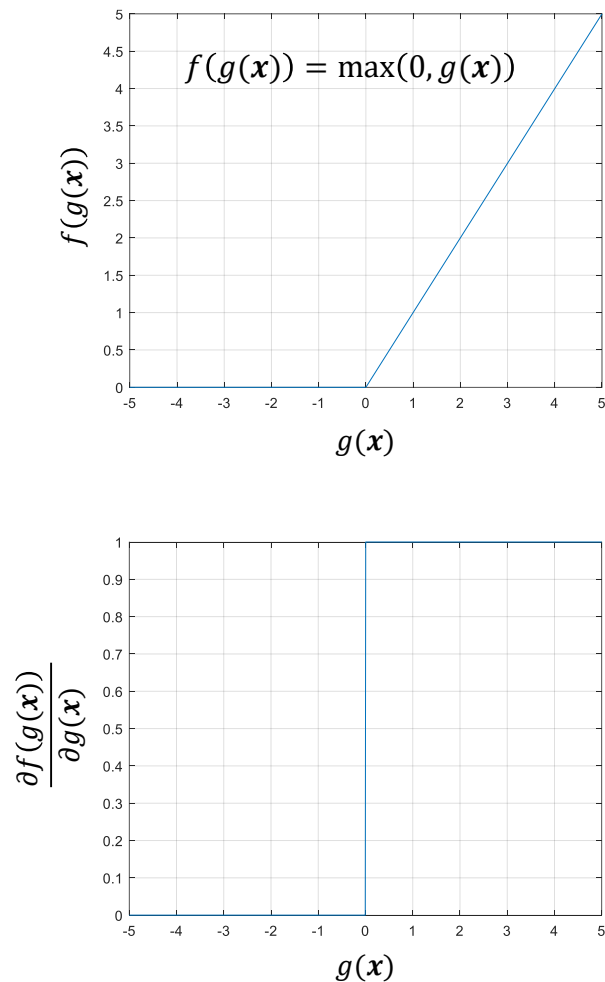
Funções de ativação: ReLU



$$\hat{y} = f(g(x)) = \max(0, g(x))$$

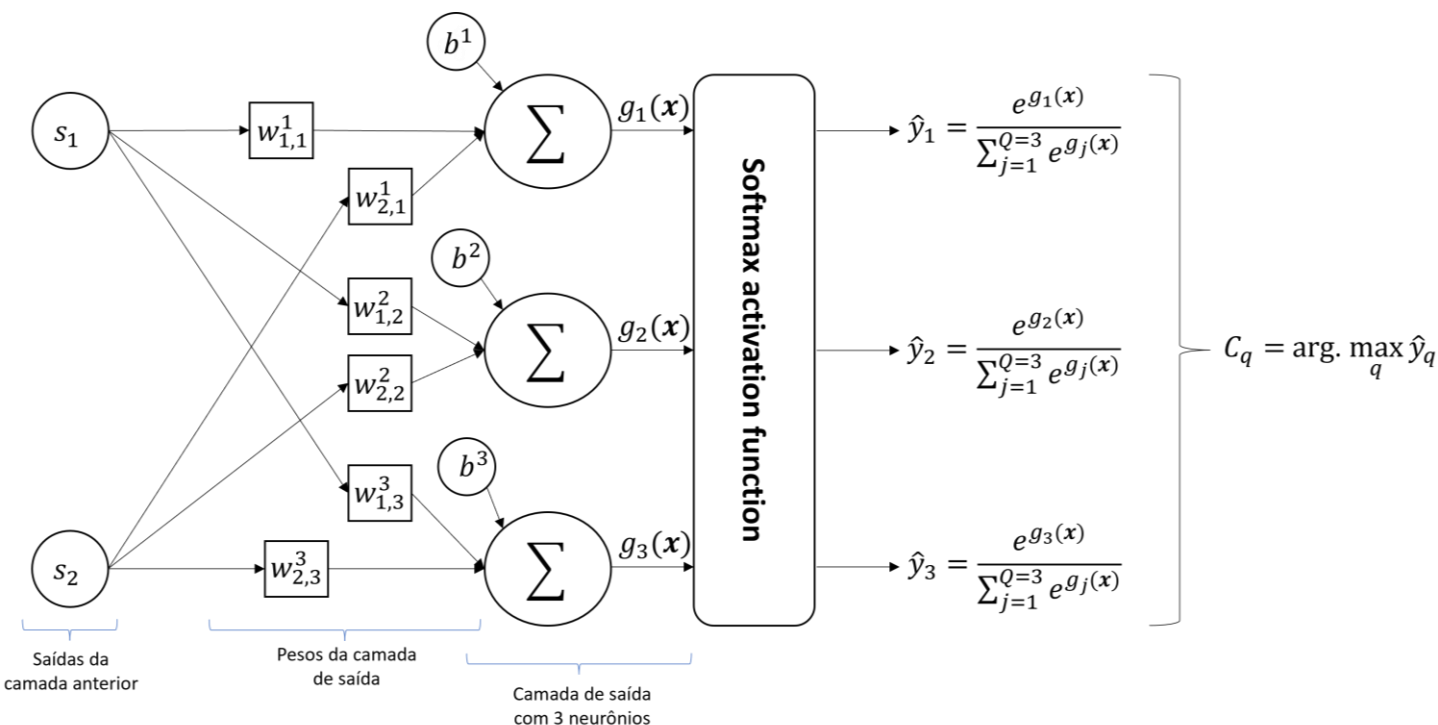
- Ajuda a **mitigar o problema do desaparecimento de gradiente**, que ocorre com funções de ativação como a sigmoide e tangente hiperbólica quando usadas **em redes profundas**.
- Ela introduz **não linearidade** nas redes, permitindo que elas **aprendam padrões e a modelar relacionamentos complexos nos dados**.
- **OBS.:** Redes com neurônios usando **ativação linear só podem modelar relacionamentos lineares** entre entradas e saídas.

Funções de ativação: ReLU



- Uma desvantagem é que ela pode levar ao problema da **ReLU agonizante**.
- Esse problema ocorre quando os **neurônios têm saída igual a zero durante o treinamento e, conseqüentemente, não conseguem atualizar seus pesos durante a retropropagação**.
- Para mitigar esse problema, foram propostas algumas variantes como: Leaky ReLU, Parametric ReLU, Exponential Linear Units (ELU) e Gaussian Error Linear Unit (GELU).

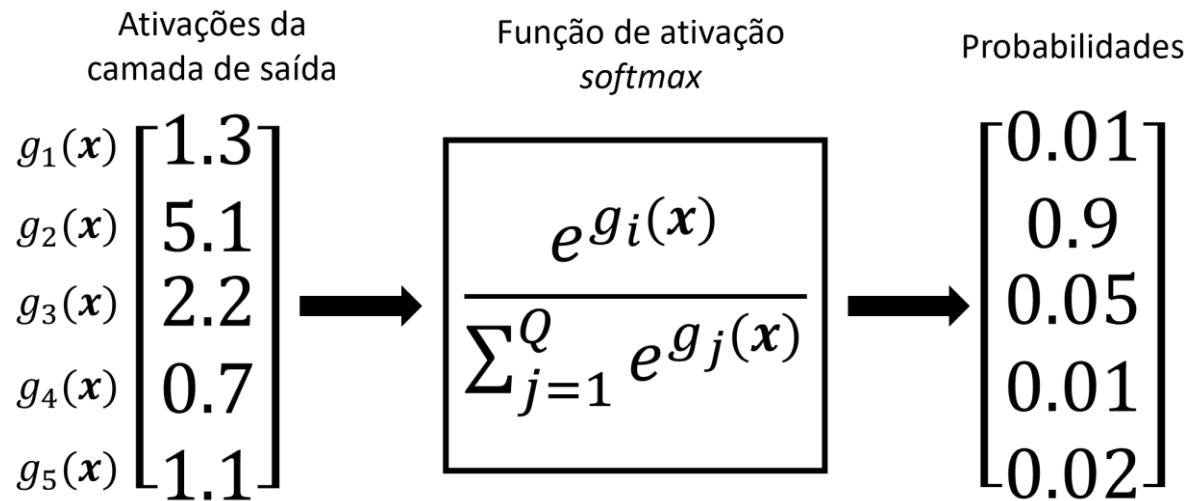
Funções de ativação: Softmax



$Q = 3$ (número de classes)

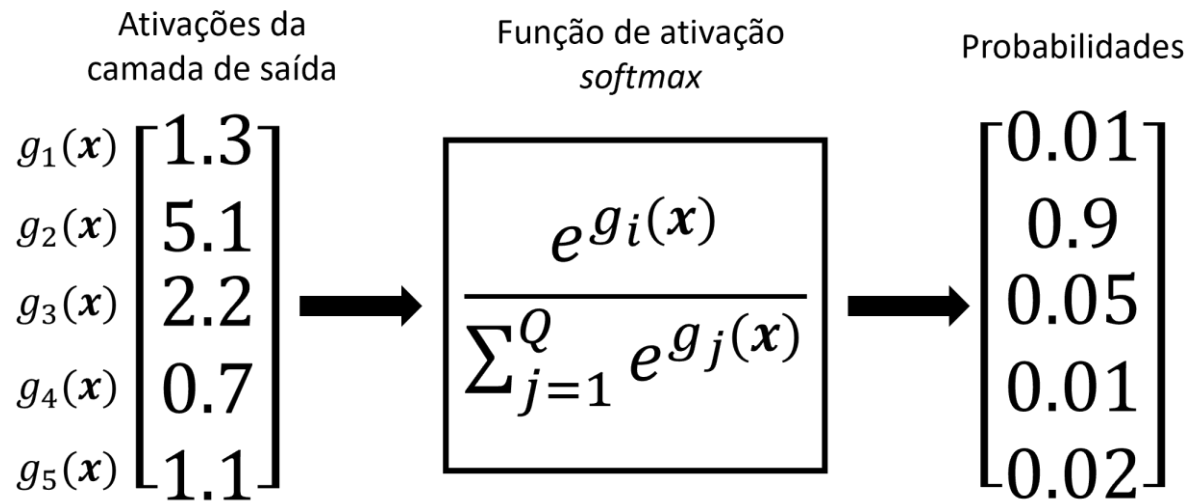
- Usada frequentemente na **camada de saída** de **redes neurais** empregadas em tarefas de **classificação**, onde a rede precisa **atribuir** uma **probabilidade para cada classe** possível.
- Transforma um **vetor de valores de ativação em uma distribuição de probabilidades**.
 - Normaliza as ativações.
- É uma generalização da função sigmoide.

Funções de ativação: Softmax



- A função opera da seguinte maneira.
- Dado um **vetor de valores de ativação**, $g_i(x)$, $\forall i$, ou seja, a soma ponderada das entradas de um neurônio, ela
 - calcula a exponencial de cada ativação,
 - normaliza essas exponenciais dividindo-as pela soma de todas as exponenciais
 - e, por final, gera uma distribuição de probabilidades das classes.

Funções de ativação: Softmax



- Ela **garante que as saídas estejam no intervalo entre 0 e 1, e a soma de todas as saídas seja igual a 1**, tornando-as interpretáveis como uma **função massa de probabilidade**.
- Isso é crucial para tarefas de classificação **multiclasse**, onde se deseja determinar a probabilidade de um exemplo de entrada pertencer a cada classe possível.
 - **Multiclasse**: cada **exemplo** é atribuído a uma única **classe exclusiva**.

Compilando o modelo

```
model.compile(optimizer = 'adam',  
              loss = 'sparse_categorical_crossentropy',  
              metrics = ['accuracy']  
              )  
  
history = model.fit(X_train, y_train, epochs=20)
```

- Vamos usar o otimizador Adam, o qual *ajusta adaptativamente passos de aprendizagem independentes* dependendo do *histórico de gradientes passados*, *acelerando a convergência do algoritmo*.

Compilando o modelo

```
model.compile(optimizer = 'adam',  
              loss = 'sparse_categorical_crossentropy',  
              metrics = ['accuracy']  
              )  
  
history = model.fit(X_train, y_train, epochs=20)
```

- Como função de erro (i.e., *loss*), usaremos a função conhecida como *entropia cruzada categórica*.
- *Não usamos o MSE* pois ele é uma *métrica usada em problemas de regressão, onde queremos prever valores contínuos*.

Compilando o modelo

```
model.compile(optimizer = 'adam',  
              loss = 'sparse_categorical_crossentropy',  
              metrics = ['accuracy']  
              )  
  
history = model.fit(X_train, y_train, epochs=20)
```

- No entanto, em *problemas de classificação*, as *saídas são categóricas*, ou seja, pertencem a um conjunto discreto e finito de classes.
- As saídas podem ser *codificadas* como *valores inteiros* ou *vetores one-hot*.
- Nesses casos, usamos como função de erro a entropia cruzada categórica ou sua versão esparsa.

Entropia cruzada categórica

- Métrica frequentemente utilizada para avaliar o **desempenho de modelos de classificação multiclasse**.
- Ela mede a **diferença entre as distribuições de probabilidade preditas pelo modelo e as distribuições dos rótulos** das classes.
- A função da **entropia cruzada média** é dada por

$$J_e(\mathbf{W}) = -\frac{1}{N} \sum_{n=0}^{N-1} \sum_{q=0}^{Q-1} 1\{y(n) == q\} \log \left(\hat{y}_q(n) \right),$$

O erro tende a 0 quando $\hat{y}_q(n)$ tende a 1, caso contrário, o erro aumenta.

onde Q é o número classes, N é o número de exemplos, $1\{\cdot\}$ é a **função indicadora** ($1\{\text{condição verdadeira}\} = 1$ e $1\{\text{condição falsa}\} = 0$), \mathbf{W} é a matriz de **pesos**, $y(n)$ é rótulo do n -ésimo exemplo de entrada e $\hat{y}_q(n)$ é a q -ésima saída do modelo para o n -ésimo exemplo de entrada.

Entropia cruzada categórica

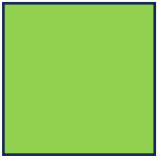

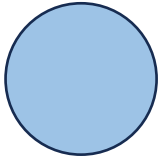
- Usando-se a codificação **one-hot**, a equação pode ser re-escrita como

$$J_e(\mathbf{W}) = -\frac{1}{N} \sum_{n=0}^{N-1} \mathbf{y}(n)^T \log(\hat{\mathbf{y}}(n)),$$

onde $\mathbf{y}(n) = [1\{y(n) == 0\}, \dots, 1\{y(n) == Q - 1\}]^T \in \mathbb{R}^{Q \times 1}$ é um vetor coluna com os rótulos utilizando a codificação **one-hot** e $\hat{\mathbf{y}}(n) = [\hat{y}_0(n), \dots, \hat{y}_{Q-1}(n)]^T \in \mathbb{R}^{Q \times 1}$ é um vetor coluna com as Q saídas do modelo para o n -ésimo exemplo de entrada.

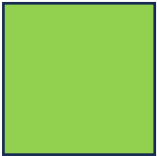

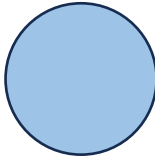
- Durante o **treinamento** do modelo, o **objetivo é minimizar a entropia cruzada categórica**, ou seja, fazer com que as **probabilidades previstas pelo modelo se aproximem ao máximo dos rótulos dos exemplos**.

Entropia cruzada categórica esparsa

Classe			
Rótulo	0	1	2
Codificação <i>one-hot</i>	[1,0,0]	[0,1,0]	[0,0,1]

- É uma variação da entropia cruzada categórica que **ao invés de usar a codificação one-hot para os rótulos, utiliza valores inteiros** para indicar a classe a que pertence um exemplo.
- Essa variação é particularmente **útil quando se lida com uma grande quantidade de classes**.
- Por exemplo, imaginem armazenar os vetores **one-hot** de um problema com mil ou mais classes (e.g., ImageNet-1K, ImageNet-21K).

Entropia cruzada categórica esparsa

Classe			
Rótulo	0	1	2
Codificação one-hot	[1,0,0]	[0,1,0]	[0,0,1]

- Isso ***economiza memória***, já que apenas ***um valor inteiro é necessário em vez de um vetor one-hot***.
- Além disso, há uma ***diminuição no tempo de cálculo***, já que podemos ***indexar diretamente a probabilidade de saída correta através do índice fornecido pelo rótulo***.
 - Com isso, evita-se vários cálculos de multiplicação, log e soma.

Compilando o modelo

```
model.compile(optimizer = 'adam',  
              loss = 'sparse_categorical_crossentropy',  
              metrics = ['accuracy']  
)  
  
history = model.fit(X_train, y_train, epochs=20)
```

- Se os rótulos são valores *inteiros*, usamos a função de perda *sparse_categorical_crossentropy*.
- Caso contrário, se os rótulos estiverem no formato *one-hot*, deve-se usar a função de perda *categorical_crossentropy*.

Treinando o modelo

```
model.compile(optimizer = 'adam',  
              loss = 'sparse_categorical_crossentropy',  
              metrics = ['accuracy']  
)
```

```
history = model.fit(X_train, y_train, epochs=20)
```

- Treinamos o modelo por apenas 20 épocas.
 - Veremos no exemplo que com essa quantidade de épocas já teremos bons resultados.

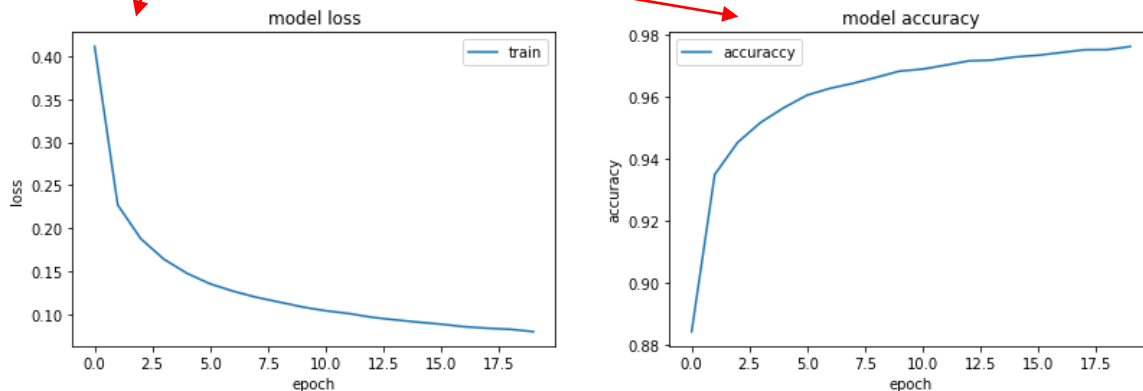


Treinar
modelo

Avaliando o modelo

```
model.compile(optimizer = 'adam',  
              loss = 'sparse_categorical_crossentropy',  
              metrics = ['accuracy']  
)
```

```
history = model.fit(X_train, y_train, epochs=20)
```



Avaliar e
otimizar

- Na sequência, avaliamos o desempenho do modelo durante o treinamento.
- Além da entropia cruzada, vejam que usamos a **acurácia** como métrica.
 - Acurácia mede a proporção de exemplos classificados corretamente em relação ao número total de exemplos.
- **OBS.:** A **função minimizada é a da entropia cruzada**. A acurácia é apenas calculada após cada época e **não influencia no treinamento**.

Realizando inferências

```
classifications = model.predict(X_test)

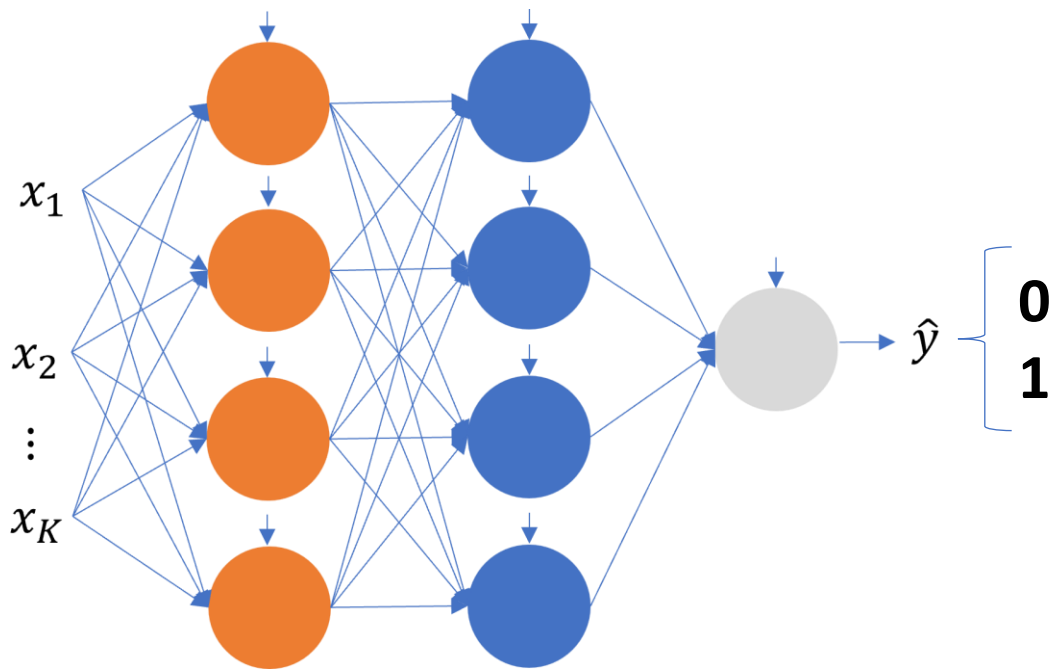
print(classifications[0])
# retorna o índice da maior probabilidade.
print("Predicted class: ", np.argmax(classifications[0]))
print("The actual class: ", y_test[0])
```

- Por fim, fazemos *inferências* com o modelo treinado usando com *amostras inéditas* para avaliar sua *capacidade de generalização*.



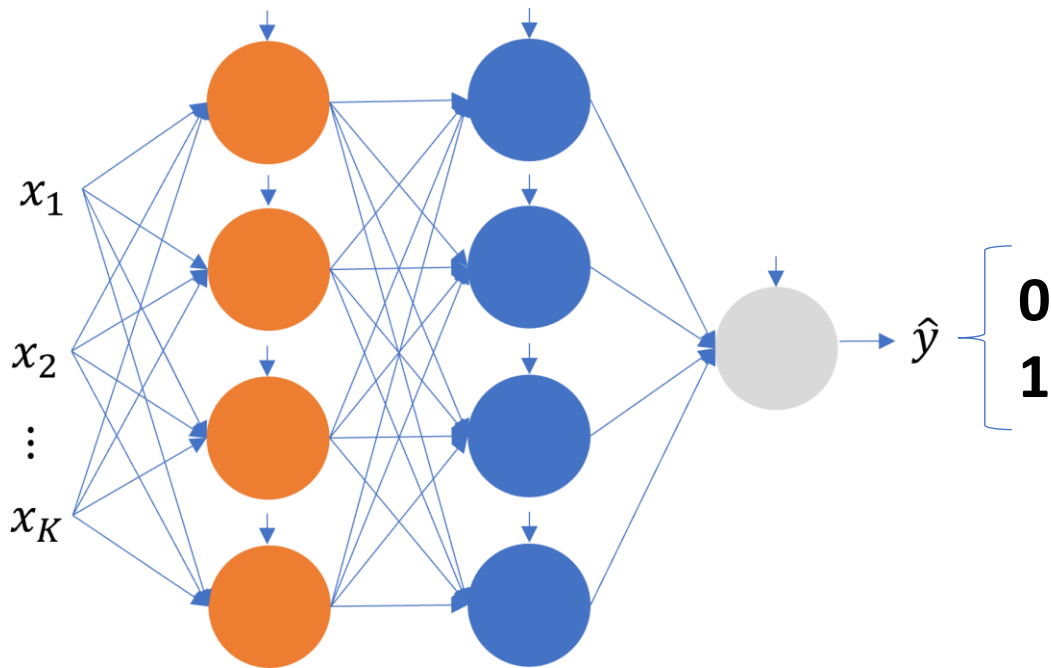
Inferência

Classificação binária



- No caso da **classificação binária**, se os rótulos forem escalares inteiros, com valor 0 ou 1, podemos criar um **classificador binário** com **apenas uma saída**.
- Ou seja, com apenas **um neurônio na camada de saída**.
- Nesse caso, normalmente, também usamos funções ReLU como ativação das camadas ocultas.
- Porém, nesse caso a **função sigmoide é usada como ativação da camada de saída**.

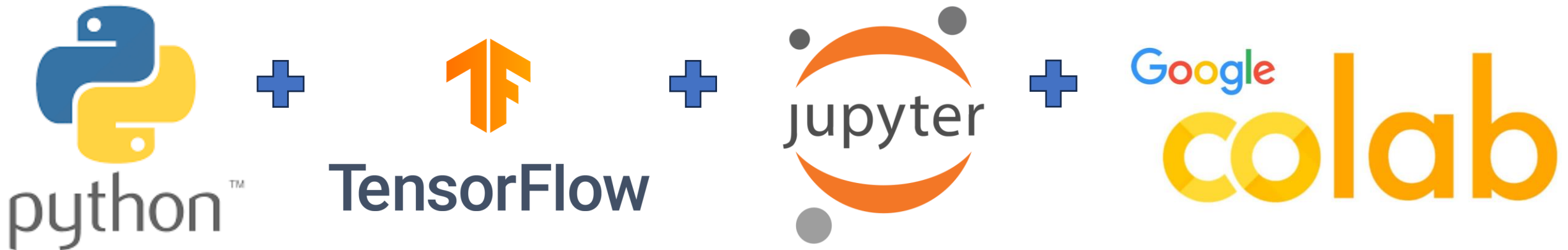
Classificação binária



- O otimizador é em geral escolhido entre Adam ou SGD, mas existem outros que podem ser testados: [Nadam, RMSProp, etc.](#)
- A função de perda deve ser a da **entropia cruzada binária**.
 - No TensorFlow ela é implementada pela classe [BinaryCrossentropy](#).
- Se estivermos lidando com imagens e a rede for do tipo **dense feed forward**, devemos usar uma camada **Flatten** na entrada.

Exemplo

- [Exemplo: Classificação de imagens com dígitos escritos à mão.](#)



Atividades

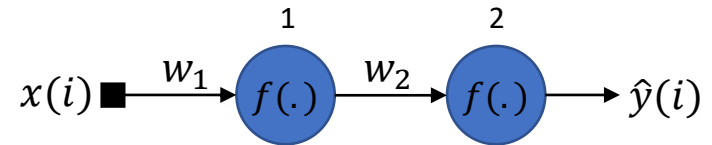
- Quiz: “***TP557 – Classificação com DNNs***”.
- Exercício: [Detecção de peças de roupa](#)

Perguntas?

Obrigado!

Anexo I: Dissipação do gradiente

Dissipação do Gradiente



Considerações:

- 2 x neurônios com função de ativação sigmoide, $f(\cdot)$.
- $g_1 = xw_1 \rightarrow$ entrada (i.e., ativação) do primeiro neurônio.
- $z_1 = f(xw_1) \rightarrow$ saída do primeiro neurônio.
- $g_2 = z_1w_2 = f(xw_1)w_2 \rightarrow$ entrada (i.e., ativação) do segundo neurônio.
- $\hat{y} = f(f(xw_1)w_2) \rightarrow$ saída do segundo neurônio.
- **Objetivo:** minimizar o erro quadrático médio, $J_e = \frac{1}{N} \sum_{i=1}^N (\hat{y}(i) - y(i))^2$.

Dissipação do Gradiente

- As **regras de atualização** dos dois pesos são dadas por

$$w_2 = w_2 - \alpha \frac{\partial J_e}{\partial w_2},$$

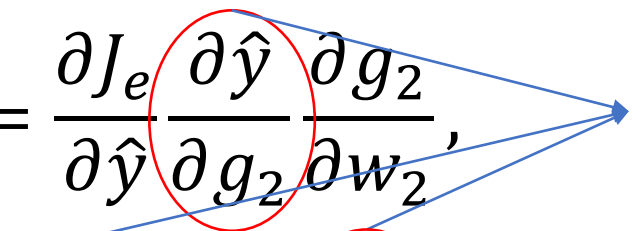
$$w_1 = w_1 - \alpha \frac{\partial J_e}{\partial w_1}.$$

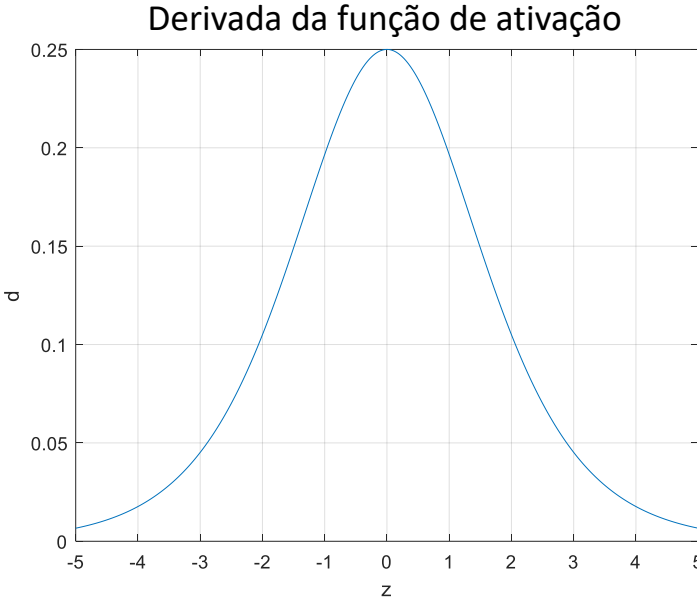
- Usando a regra da cadeia, obtemos as derivadas $\frac{\partial J_e}{\partial w_1}$ e $\frac{\partial J_e}{\partial w_2}$

$$\frac{\partial J_e}{\partial w_2} = \frac{\partial J_e}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial g_2} \frac{\partial g_2}{\partial w_2},$$

$$\frac{\partial J_e}{\partial w_1} = \frac{\partial J_e}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial g_2} \frac{\partial g_2}{\partial z_1} \frac{\partial z_1}{\partial g_1} \frac{\partial g_1}{\partial w_1}.$$

Dissipação do Gradiente

$$\frac{\partial J_e}{\partial w_2} = \frac{\partial J_e}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial g_2} \frac{\partial g_2}{\partial w_2},$$
$$\frac{\partial J_e}{\partial w_1} = \frac{\partial J_e}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial g_2} \frac{\partial g_2}{\partial z_1} \frac{\partial z_1}{\partial g_1} \frac{\partial g_1}{\partial w_1}.$$




- A derivada da função sigmoide é no máximo igual a 0.25.
- Assim, por exemplo, a primeira camada de uma rede neural com M camadas, terá as derivadas parciais da função de erro em relação a seus pesos compostas pela multiplicação de M termos no máximo iguais a 0.25.
- Isso faz com que as primeiras camadas aprendam lentamente ou nem aprendam, pois têm derivadas muito pequenas, tendendo a zero.

Anexo II: Adam

Adaptive Moment Estimation (Adam)

Um passo adaptativo para cada elemento do vetor gradiente.

$$a \leftarrow a - \frac{\alpha_0}{\sqrt{\hat{v}} + \varepsilon} \hat{m}$$

$$\hat{m} = m / (1 - \beta_1^t)$$

$$\hat{v} = v / (1 - \beta_2^t)$$

ε é um valor pequeno, e.g., 1e-8, que garante que não ocorra divisão por zero.

Estimativa do momento de primeira ordem (média).

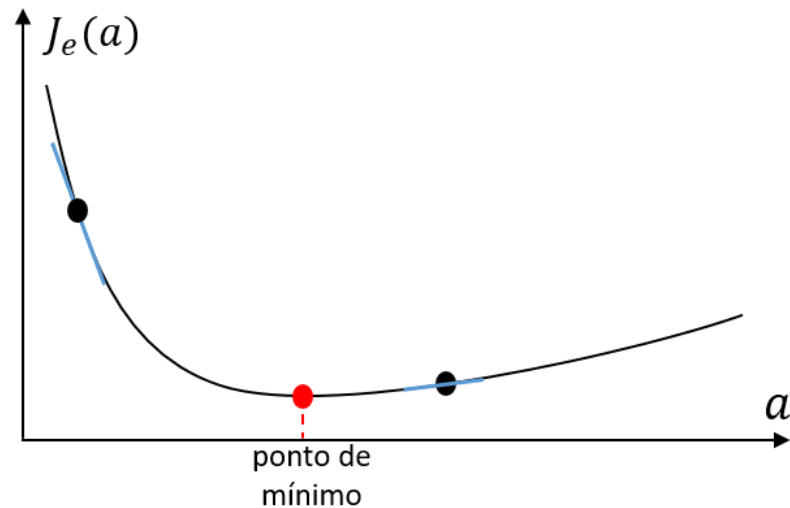
$$m \leftarrow \beta_1 m + (1 - \beta_1) \nabla J_e(a)$$

$$v \leftarrow \beta_2 v + (1 - \beta_2) [\nabla J_e(a)]^2$$

Estimativa do momento de segunda ordem (variância não-centralizada).

- Usa **médias móveis exponencialmente decrescentes** dos gradientes anteriores e de seus quadrados para atualizar os pesos do modelo.
- A média dos gradientes anteriores suaviza as atualizações e acelera a convergência, principalmente quando usado com **estimativas estocásticas do vetor gradiente**.

Adaptive Moment Estimation (Adam)



- Em regiões íngremes, reduz rapidamente o passo de aprendizagem, pois os valores das derivadas (taxa de variação) são grandes.
 - Isso evita que ultrapasse o mínimo.
- Em regiões planas ou quase, diminui o passo de aprendizagem muito lentamente, pois as derivadas são muito pequenas.
 - Passo permanece com valor grande, fazendo com que se caminhe mais rapidamente por essas regiões.

Adaptive Moment Estimation (Adam)

$$\begin{aligned} a &\leftarrow a - \frac{\alpha_0}{\sqrt{\hat{v} + \varepsilon}} \hat{m} \\ \hat{m} &= m / (1 - \beta_1^t) \\ \hat{v} &= v / (1 - \beta_2^t) \\ \text{Termo Momentum } m &\leftarrow \beta_1 m + (1 - \beta_1) \nabla J_e(a) \\ \text{RMSProp } v &\leftarrow \beta_2 v + (1 - \beta_2) [\nabla J_e(a)]^2 \end{aligned}$$

Corrigem o viés em direção a zero devido a inicialização igual a zero das estimativas dos momentos, m e v .

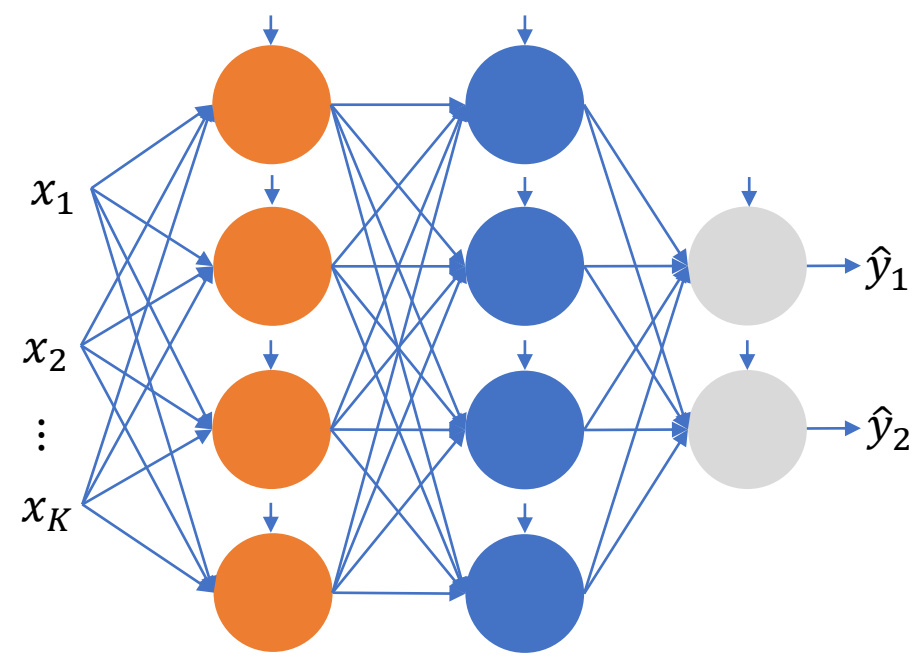
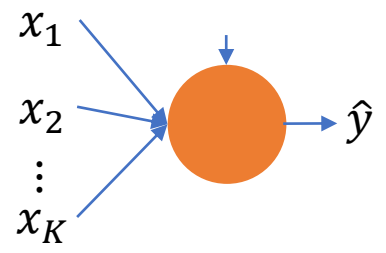
Estimativa do momento de primeira ordem (média).

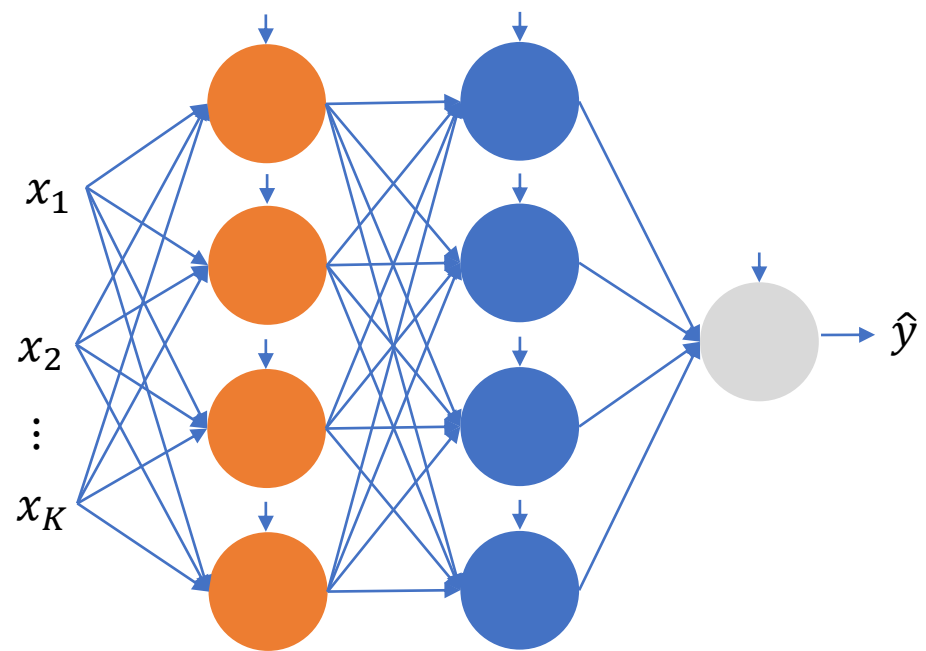
Estimativa do momento de segunda ordem (variância não-centralizada).

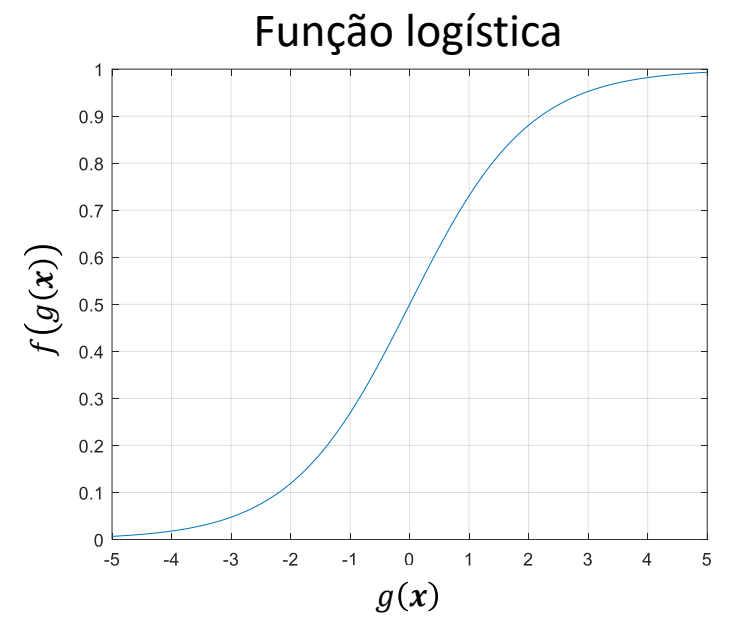
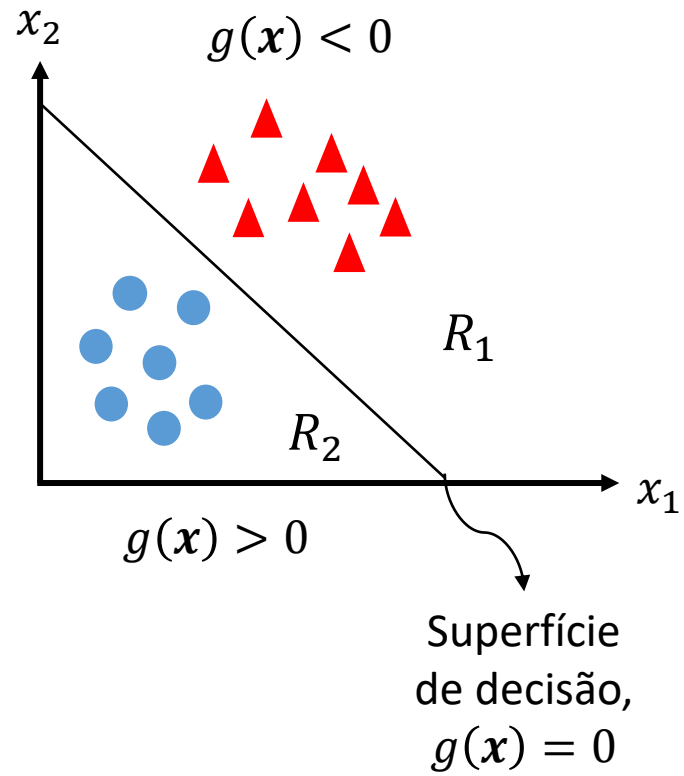
- Combinação do RMSProp com o termo momentum.
- Usa **médias móveis exponencialmente decrescentes** dos gradientes anteriores e dos quadrados dos gradientes anteriores para atualizar os pesos do modelo.
- A média dos gradientes anteriores suaviza as atualizações e acelera a convergência, principalmente quando usado com **estimativas estocásticas do vetor gradiente**.
- Bem mais robusto que os anteriores quanto à escolha dos hiperparâmetros.

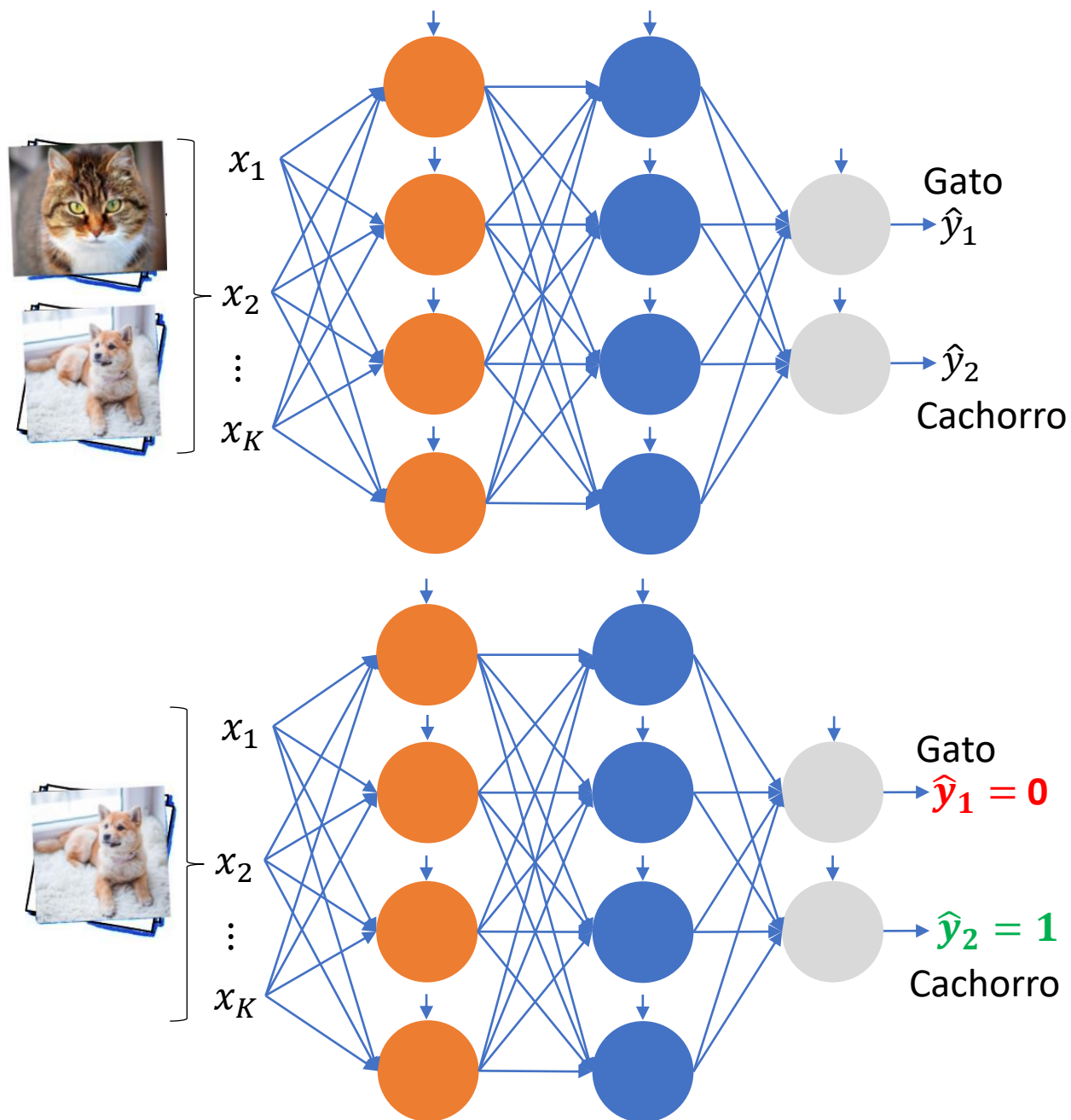
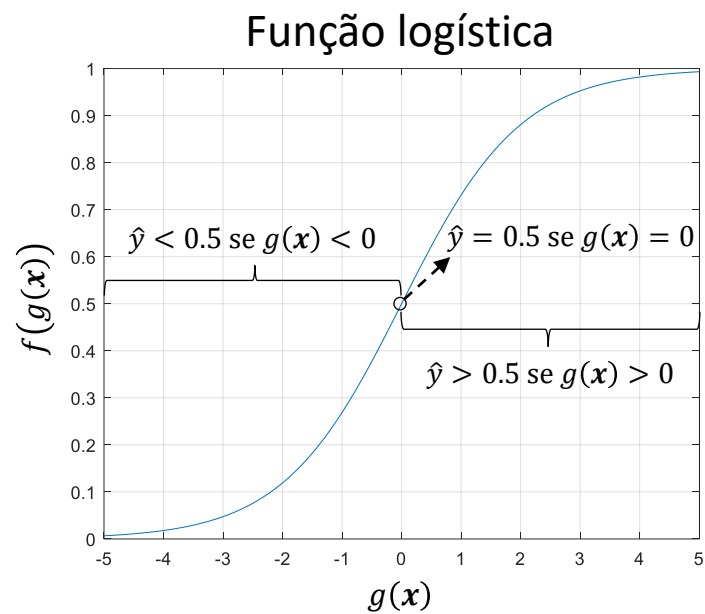
Figuras

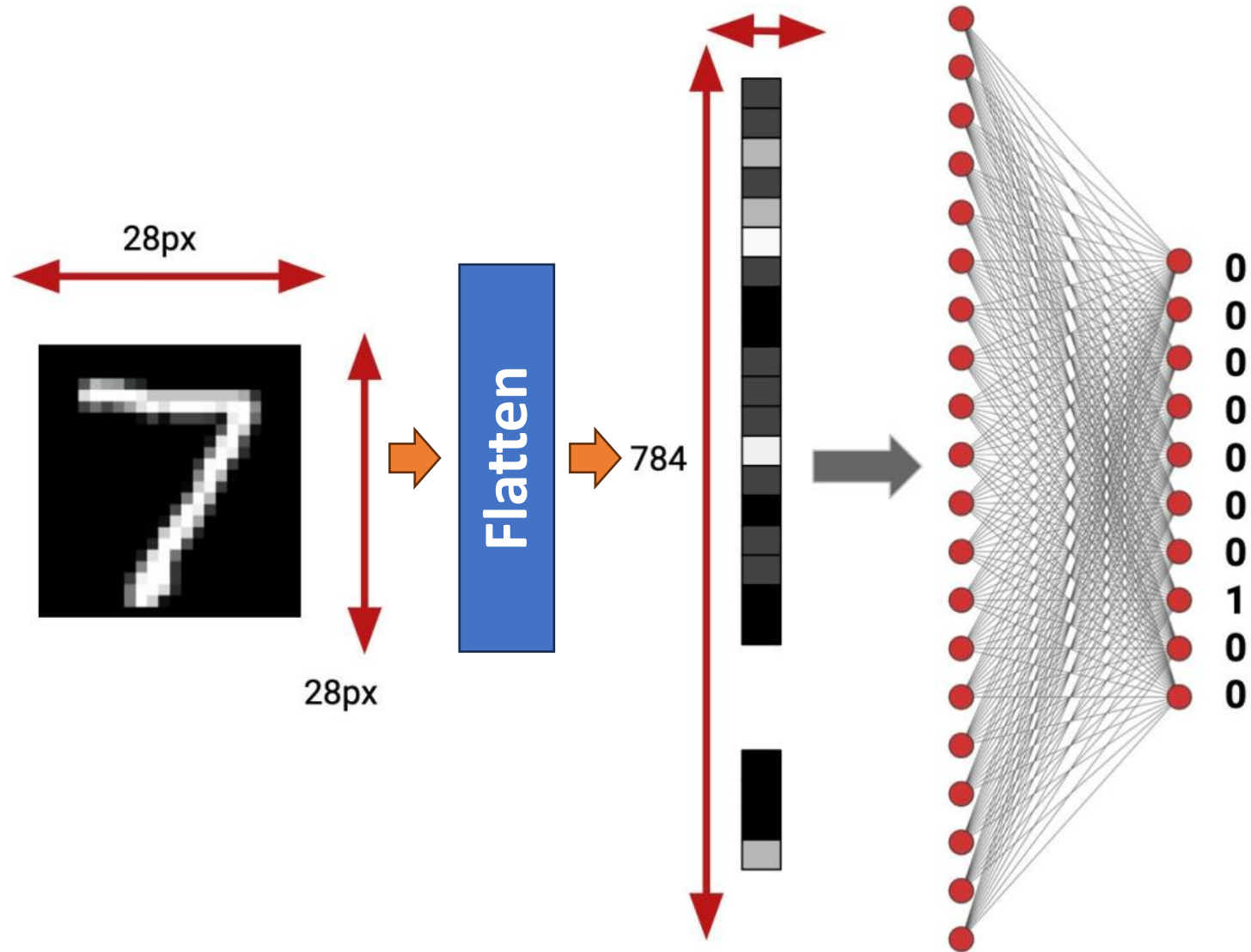


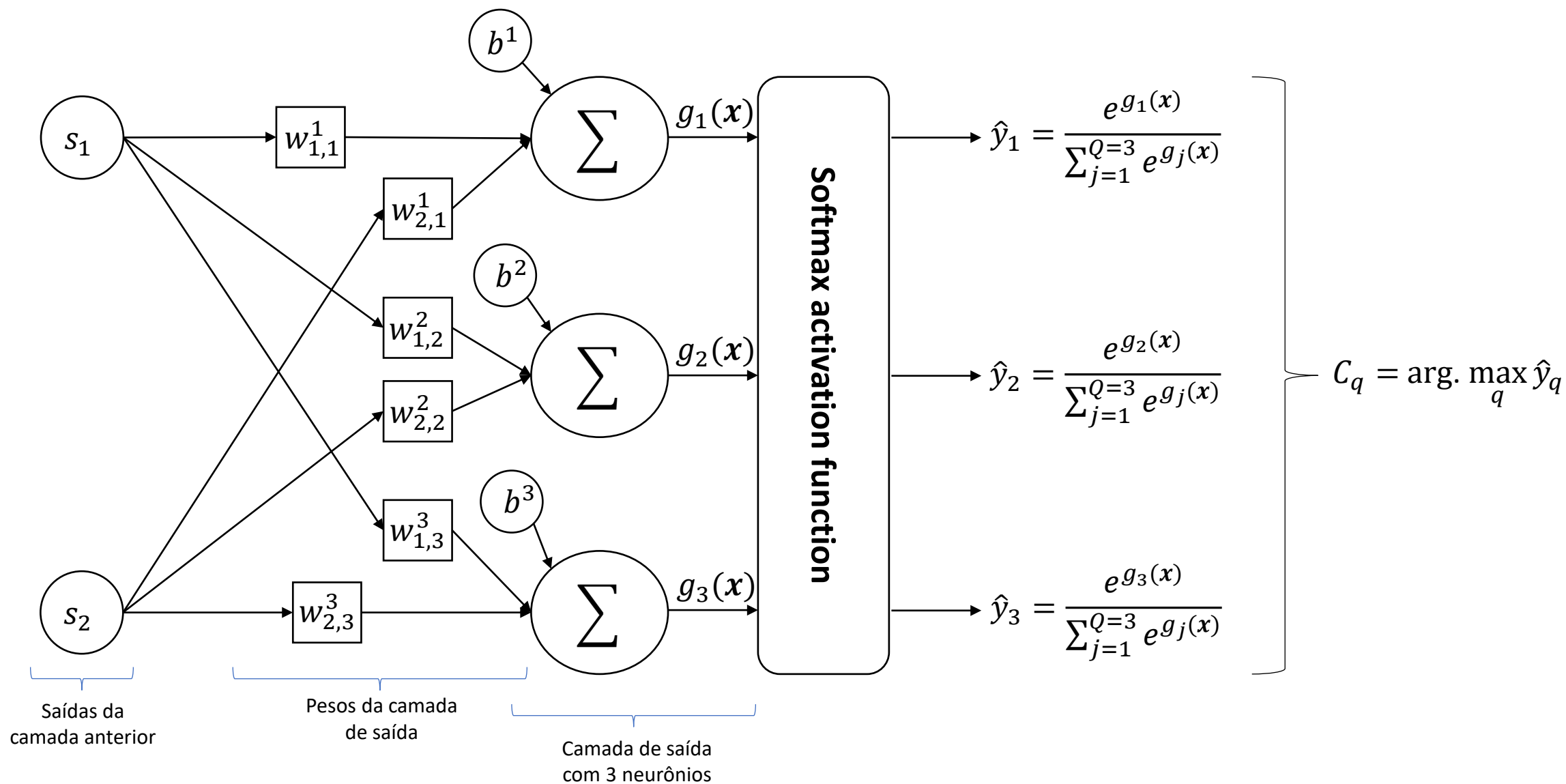












Ativações da
camada de saída

$$\begin{matrix} g_1(x) \\ g_2(x) \\ g_3(x) \\ g_4(x) \\ g_5(x) \end{matrix} \begin{bmatrix} 1.3 \\ 5.1 \\ 2.2 \\ 0.7 \\ 1.1 \end{bmatrix}$$

Função de ativação
softmax

$$\frac{e^{g_i(x)}}{\sum_{j=1}^Q e^{g_j(x)}}$$

Probabilidades

$$\begin{bmatrix} 0.01 \\ 0.9 \\ 0.05 \\ 0.01 \\ 0.02 \end{bmatrix}$$

