# Lab 2 - BCC406

## REDES NEURAIS E APRENDIZAGEM EM PROFUNDIDADE

## Regressão Logística e Rede Neural

### Prof. Eduardo e Prof. Pedro

Objetivos:

- Regressão e Descida do Gradiente

Data da entrega : 26/08

- Complete o código (marcado com ToDo) e quando requisitado, escreva textos diretamente nos notebooks. Onde tiver *None*, substitua pelo seu código.
- Execute todo notebook e salve tudo em um PDF **nomeado** como "NomeSobrenome-LabX.pdf"
- Envie o PDF via google FORM

# Classificação utilizando frameworks

- Trabalharemos com um problema de classificação: é um gato ou não é um gato.
- Utilizaremos um framework: o Tensorflow/Keras.

# Importando os pacotes

Primeiro, vamos executar a célula abaixo para importar todos os pacotes que precisaremos.

- numpy é o pacote fundamental para a computação científica com Python.
- h5py é um pacote comum para interagir com um conjunto de dados armazenado em um arquivo H5.
- matplotlib é uma biblioteca famosa para plotar gráficos em Python.
- PIL e scipy são usados aqui para testar seu modelo.
- tensorflow é uma biblioteca famosa para criar e treinar modelos de Deep Learning.
- np.random.seed (1) é usado para manter todas as chamadas de funções aleatórias.

```
import numpy as np
import h5py
import matplotlib.pyplot as plt
import random
```

```
from sklearn.metrics import accuracy_score

import tensorflow as tf
from tensorflow.python.keras import Sequential
from tensorflow.python.keras.layers import Dense
from tensorflow.keras import initializers
```

Configurando o matplotlib e a geração de dados aleatórios

```
%matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2

np.random.seed(1)
```

## Configurando o Google Colab

```
# Para Google Colab: Você vai precisar fazer o upload dos arquivos no seu drive e montá-lo
# não se esqueça de ajustar o path para o seu drive
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

## Carregando os dados (10pt)

Coloque os arquivos *train_catvnonvat.h5* e *test_catvnoncat.h5* na pasta raiz do seu Drive.
Ambos os arquivos estão na pasta dataset da pasta compartilhada.

```
# Lendo os dados (gato/não-gato)
def load_dataset():

  train_dataset = h5py.File('train_catvnoncat.h5', "r")
  train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # your train set features
  train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # your train set labels

  test_dataset = h5py.File('test_catvnoncat.h5', "r")
  test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # your test set features
  test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # your test set labels

  classes = np.array(test_dataset["list_classes"][:]) # the list of classes
```

```
    train_set_y_orig = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))
    test_set_y_orig = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))

    return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig, classes


# Lendo os dados (gato/não-gato)
treino_x_orig, treino_y, teste_x_orig, teste_y, classes = load_dataset()
```

## Pré-processamento dos dados

Pre-processamento necessário. Iremos converter a imagem 3D (64x64x3) em um único vetor 1D (12288 = 64x64x3). A figura mostra um exemplo do pré-processamento executado (imagem vetorizada)

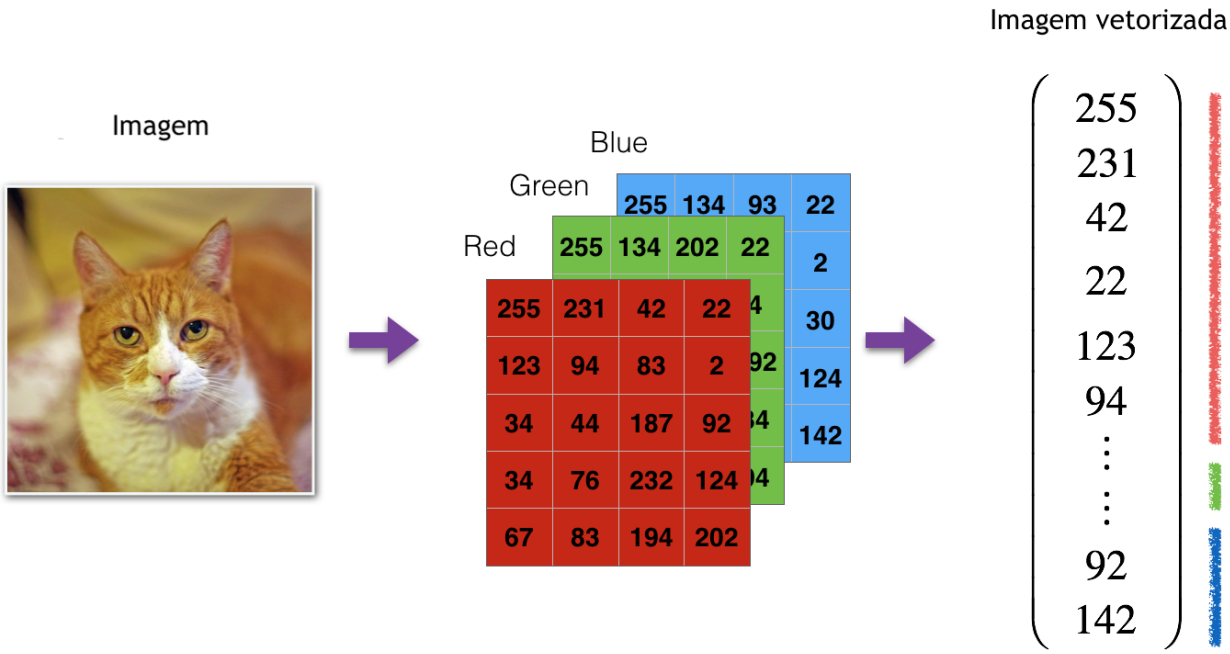



Figura: Vetorização de uma imagem.

## **ToDo:** Vetorização da imagem (10pt)

Converta as imagens do formato 64x64x3 para 12288x1.

```
m_treino = len(treino_x_orig)
m_teste = len(teste_x_orig)
num_px = teste_x_orig[1].shape[1]


#  Vetorizando as imagens de treinamento e teste

### Início do código ###
```

```
treino_x = treino_x_orig.reshape((m_treino, num_px*num_px*3)) # dica : utilize reshape par
### Fim do código ###

### Início do código ###
# Normalize os dados para ter valores de recurso entre 0 e 1.
teste_x = teste_x_orig.reshape((m_teste, num_px*num_px*3)) # dica : utilize reshape para m
### Fim do código ###
```

## Testando redes neurais e *sigmoid*

Para classificação de classes 0 ou 1, pode-se ter um único neurônio de saída e deve-se usar a operação sigmoid antes de se calcular o custo (mean-squared error ou binary cross entropy).

## Testando uma rede com uma camada oculta

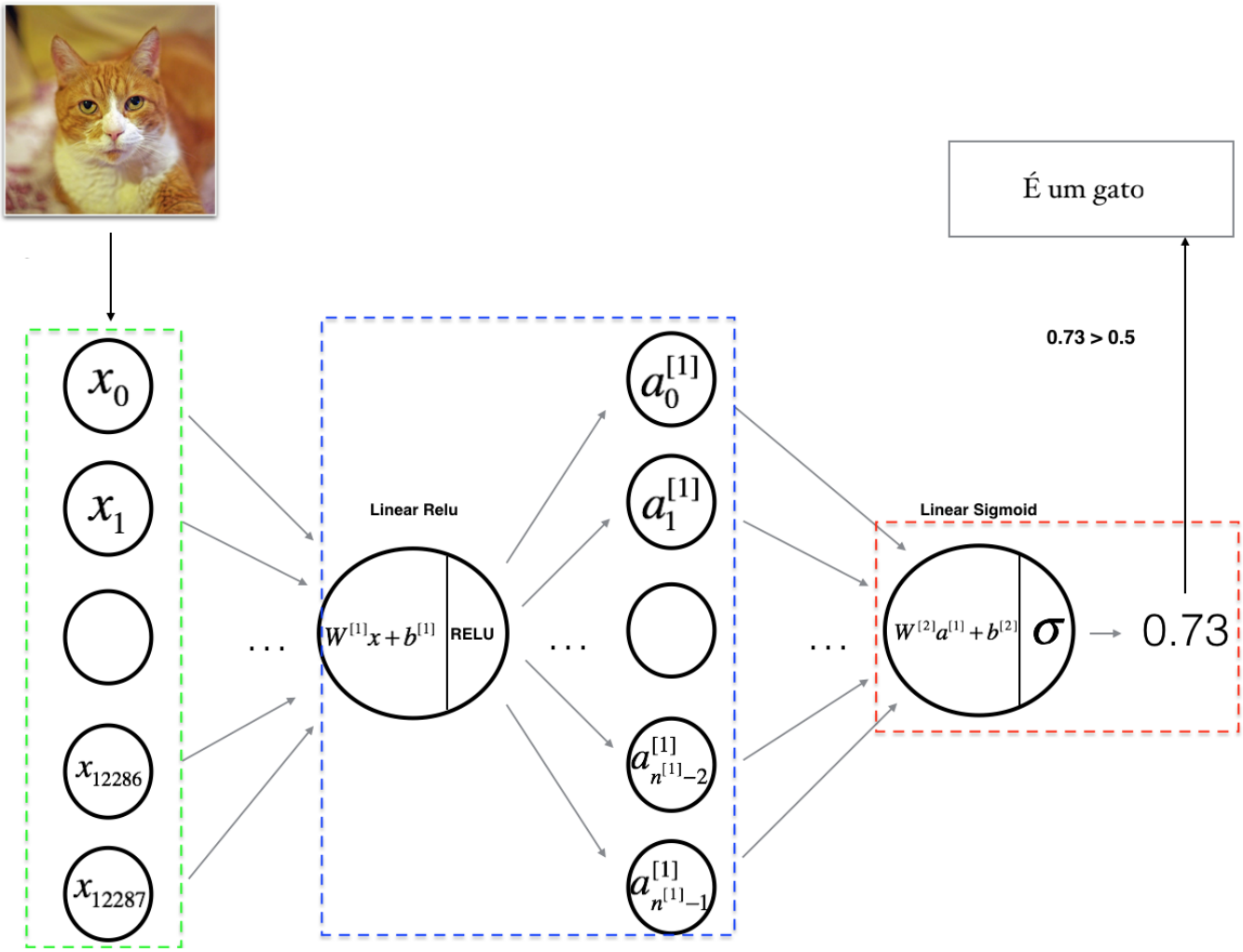



Figura 7: Rede neural com 2 camadas.
Resumo do modelo: ***ENTRADA -> LINEAR -> RELU -> LINEAR -> SIGMOID -> SAIDA***.

```
## Setando a seed
np.random.seed(1)
random.seed(1)
```

```
### Executar uma rede de 1 camada oculta ###
# Camadas da rede = [12288, 200, 1]

# Definição do modelo
model = Sequential()
model.add(Dense(200, input_shape=(12288,), activation='relu', name='CamadaOculta'))
model.add(Dense(1, activation='sigmoid', name='CamadaClassificacao'))

# Compilando o modelo
model.compile(loss='binary_crossentropy', optimizer='adam')

# Imprimindo a arquitetura da rede proposta
model.summary()

# Treinando o modelo
model.fit(treino_x, treino_y.reshape(-1), epochs=100)
```

```
7/7 [==============================] - 0s 18ms/step - loss: 1.5197e-06
Epoch 73/100
7/7 [==============================] - 0s 18ms/step - loss: 1.5074e-06
Epoch 74/100
7/7 [==============================] - 0s 17ms/step - loss: 1.4961e-06
Epoch 75/100
7/7 [==============================] - 0s 17ms/step - loss: 1.4910e-06
Epoch 76/100
7/7 [==============================] - 0s 16ms/step - loss: 1.4879e-06
Epoch 77/100
7/7 [==============================] - 0s 19ms/step - loss: 1.4847e-06
Epoch 78/100
7/7 [==============================] - 0s 16ms/step - loss: 1.4819e-06
Epoch 79/100
7/7 [==============================] - 0s 21ms/step - loss: 1.4794e-06
Epoch 80/100
7/7 [==============================] - 0s 21ms/step - loss: 1.4766e-06
Epoch 81/100
7/7 [==============================] - 0s 22ms/step - loss: 1.4737e-06
Epoch 82/100
7/7 [==============================] - 0s 23ms/step - loss: 1.4715e-06
Epoch 83/100
7/7 [==============================] - 0s 22ms/step - loss: 1.4686e-06
Epoch 84/100
7/7 [==============================] - 0s 23ms/step - loss: 1.4657e-06
Epoch 85/100
7/7 [==============================] - 0s 20ms/step - loss: 1.4644e-06
Epoch 86/100
7/7 [==============================] - 0s 16ms/step - loss: 1.4610e-06
Epoch 87/100
7/7 [==============================] - 0s 17ms/step - loss: 1.4586e-06
Epoch 88/100
7/7 [==============================] - 0s 17ms/step - loss: 1.4560e-06
Epoch 89/100
7/7 [==============================] - 0s 17ms/step - loss: 1.4540e-06
Epoch 90/100
7/7 [==============================] - 0s 18ms/step - loss: 1.4516e-06
Epoch 91/100
7/7 [==============================] - 0s 19ms/step - loss: 1.4493e-06
Epoch 92/100
7/7 [==============================] - 0s 16ms/step - loss: 1.4469e-06
Epoch 93/100
```

```
7/7 [==============================] - 0s 17ms/step - loss: 1.4445e-06
Epoch 94/100
7/7 [==============================] - 0s 17ms/step - loss: 1.4419e-06
Epoch 95/100
7/7 [==============================] - 0s 16ms/step - loss: 1.4395e-06
Epoch 96/100
7/7 [==============================] - 0s 16ms/step - loss: 1.4375e-06
Epoch 97/100
7/7 [==============================] - 0s 17ms/step - loss: 1.4341e-06
Epoch 98/100
7/7 [==============================] - 0s 16ms/step - loss: 1.4323e-06
Epoch 99/100
7/7 [==============================] - 0s 18ms/step - loss: 1.4289e-06
Epoch 100/100
7/7 [==============================] - 0s 18ms/step - loss: 1.4258e-06
<tensorflow.python.keras.callbacks.History at 0x7f29d1020690>
```

**Use os parâmetros treinados** para classificar as imagens de treinamento e teste e verificar a acurácia.

```
## Predição da rede
print(f'Acurácia no treino: {accuracy_score(treino_y.reshape(-1), np.round(model.predict(t
print(f'Acurácia no teste: {accuracy_score(teste_y.reshape(-1), np.round(model.predict(tes
```

```
Acurácia no treino: 100.00
Acurácia no teste: 68.00
```

**Resultado esperado:**

```
Acurácia treino = 100%
Acurácia teste = 64%
```

## **ToDo:** Análise dos resultados (5pt)

Por que você obteve 100% no treino e apenas 64% no teste?

Porque a rede neural ficou muito sobreajustada aos dados de treino e não conseguiu acertar bem os dados de teste, houve um overfitting.

## Testando com uma rede com três camadas ocultas

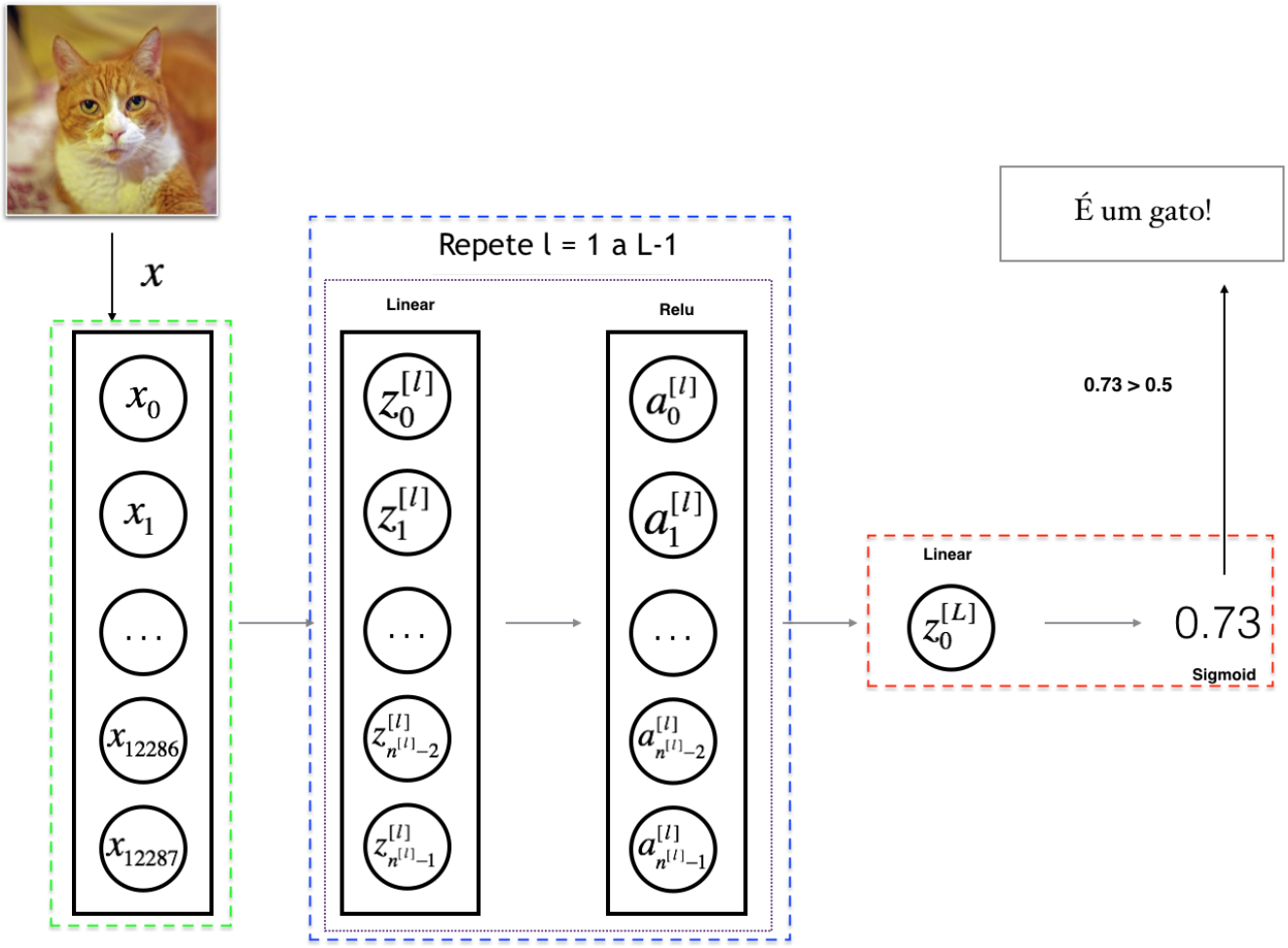



Figura 8: Rede neural com L camadas.
Resumo do modelo: ***ENTRADA -> LINEAR -> RELU -> LINEAR -> SIGMOID -> SAIDA***.

```
## Setando a seed
np.random.seed(1)
random.seed(1)

### Executar uma rede de 3 camadas ocultas ###
# Camadas da rede = [12288, 200, 70, 5, 1]

# Definição do modelo
model = Sequential()
model.add(Dense(200, input_shape=(12288,), activation='relu', name='CamadaOculta1'))
model.add(Dense(70, activation='relu', name='CamadaOculta2'))
model.add(Dense(5, activation='relu', name='CamadaOculta3'))
model.add(Dense(1, activation='sigmoid', name='CamadaClassificacao'))

# Compilando o modelo
model.compile(loss='binary_crossentropy', optimizer='adam')

# Imprimindo a arquitetura da rede proposta
model.summary()

# Treinando o modelo
model.fit(treino_x, treino_y.reshape(-1), epochs=50)
```

```
     7/7 [==============================] - 0s 19ms/step - loss: 0.6780
     Epoch 22/50
```

```
Epoch 23/50
7/7 [==============================] - 0s 20ms/step - loss: 0.6774
Epoch 24/50
7/7 [==============================] - 0s 24ms/step - loss: 0.6767
Epoch 25/50
7/7 [==============================] - 0s 20ms/step - loss: 0.6760
Epoch 26/50
7/7 [==============================] - 0s 20ms/step - loss: 0.6754
Epoch 27/50
7/7 [==============================] - 0s 19ms/step - loss: 0.6748
Epoch 28/50
7/7 [==============================] - 0s 20ms/step - loss: 0.6741
Epoch 29/50
7/7 [==============================] - 0s 20ms/step - loss: 0.6735
Epoch 30/50
7/7 [==============================] - 0s 20ms/step - loss: 0.6729
Epoch 31/50
7/7 [==============================] - 0s 21ms/step - loss: 0.6724
Epoch 32/50
7/7 [==============================] - 0s 19ms/step - loss: 0.6717
Epoch 33/50
7/7 [==============================] - 0s 19ms/step - loss: 0.6712
Epoch 34/50
7/7 [==============================] - 0s 21ms/step - loss: 0.6706
Epoch 35/50
7/7 [==============================] - 0s 21ms/step - loss: 0.6701
Epoch 36/50
7/7 [==============================] - 0s 21ms/step - loss: 0.6694
Epoch 37/50
7/7 [==============================] - 0s 19ms/step - loss: 0.6689
Epoch 38/50
7/7 [==============================] - 0s 22ms/step - loss: 0.6685
Epoch 39/50
7/7 [==============================] - 0s 24ms/step - loss: 0.6679
Epoch 40/50
7/7 [==============================] - 0s 22ms/step - loss: 0.6674
Epoch 41/50
7/7 [==============================] - 0s 23ms/step - loss: 0.6670
Epoch 42/50
7/7 [==============================] - 0s 25ms/step - loss: 0.6664
Epoch 43/50
7/7 [==============================] - 0s 22ms/step - loss: 0.6661
Epoch 44/50
7/7 [==============================] - 0s 26ms/step - loss: 0.6656
Epoch 45/50
7/7 [==============================] - 0s 24ms/step - loss: 0.6651
Epoch 46/50
7/7 [==============================] - 0s 24ms/step - loss: 0.6648
Epoch 47/50
7/7 [==============================] - 0s 23ms/step - loss: 0.6643
Epoch 48/50
7/7 [==============================] - 0s 23ms/step - loss: 0.6639
Epoch 49/50
7/7 [==============================] - 0s 24ms/step - loss: 0.6634
Epoch 50/50
7/7 [==============================] - 0s 26ms/step - loss: 0.6630
<tensorflow.python.keras.callbacks.History at 0x7f34f3070f50>
```

**Use os parâmetros treinados** para classificar as imagens de treinamento e teste e verificar a acurácia.

```
## Predição da rede
print(f'Train accuracy: {accuracy_score(treino_y.reshape(-1), np.round(model.predict(treir
print(f'Test accuracy: {accuracy_score(teste_y.reshape(-1), np.round(model.predict(teste_>
```

```
Train accuracy: 65.55
Train accuracy: 34.00
```

**Resultado esperado:**

```
Acurácia treino = 65.55%
Acurácia teste = 34.00%
```

## **ToDo:** Análise dos resultados (5pt)

O resultado com três camadas ocultas foi melhor ou pior do que usa somente uma camada? Tente explicar os motivos.

Foi pior, errou mais no conjunto de treino e de teste. Isso aconteceu porque a informação passadas pelas múltiplas camadas alteravam muito os pesos, impedindo o modelo de aprender.

## **ToDo:** Teste uma rede (20pt)

Crie uma arquitetura e treine/teste o seu modelo

```
### Início do código ###
## Setando a seed
np.random.seed(1)
random.seed(1)

### Executar uma rede de 1 camada oculta ###
# Camadas da rede = [12288, 8193, 1]

# Definição do modelo
model2 = Sequential()
model2.add(Dense(8193, input_shape=(12288,), activation='relu', name='CamadaOculta'))
model2.add(Dense(1, activation='sigmoid', name='CamadaClassificacao'))

# Compilando o modelo
model2.compile(loss='binary_crossentropy', optimizer='adam')
```

```
# Imprimindo a arquitetura da rede proposta
model2.summary()

# Treinando o modelo
model2.fit(treino_x, treino_y.reshape(-1), epochs=100)
### Fim do código ###

## Predição da rede
print(f'Train accuracy: {accuracy_score(treino_y.reshape(-1), np.round(model2.predict(trei
print(f'Test accuracy: {accuracy_score(teste_y.reshape(-1), np.round(model2.predict(teste_
```

```
Epoch 73/100
7/7 [==============================] - 5s 639ms/step - loss: 3.0609
Epoch 74/100
7/7 [==============================] - 5s 684ms/step - loss: 2.4821
Epoch 75/100
7/7 [==============================] - 5s 636ms/step - loss: 2.1131
Epoch 76/100
7/7 [==============================] - 5s 716ms/step - loss: 1.8199
Epoch 77/100
7/7 [==============================] - 4s 601ms/step - loss: 1.3447
Epoch 78/100
7/7 [==============================] - 5s 636ms/step - loss: 0.4246
Epoch 79/100
7/7 [==============================] - 5s 687ms/step - loss: 0.7032
Epoch 80/100
7/7 [==============================] - 5s 696ms/step - loss: 1.5760e-04
Epoch 81/100
7/7 [==============================] - 5s 660ms/step - loss: 0.0631
Epoch 82/100
7/7 [==============================] - 5s 715ms/step - loss: 0.0191
Epoch 83/100
7/7 [==============================] - 4s 577ms/step - loss: 0.1038
Epoch 84/100
7/7 [==============================] - 4s 584ms/step - loss: 0.0961
Epoch 85/100
7/7 [==============================] - 4s 589ms/step - loss: 0.1032
Epoch 86/100
7/7 [==============================] - 4s 603ms/step - loss: 0.1483
Epoch 87/100
7/7 [==============================] - 4s 580ms/step - loss: 0.8147
Epoch 88/100
7/7 [==============================] - 4s 588ms/step - loss: 4.1796
Epoch 89/100
7/7 [==============================] - 5s 737ms/step - loss: 8.6958
Epoch 90/100
7/7 [==============================] - 5s 649ms/step - loss: 5.6132
Epoch 91/100
7/7 [==============================] - 5s 688ms/step - loss: 2.8586
Epoch 92/100
7/7 [==============================] - 5s 675ms/step - loss: 1.3394
Epoch 93/100
7/7 [==============================] - 5s 715ms/step - loss: 0.1700
Epoch 94/100
7/7 [==============================] - 5s 743ms/step - loss: 0.2610
Epoch 95/100
7/7 [==============================] - 6s 802ms/step - loss: 0.1660
Epoch 96/100
7/7 [==============================] - 5s 690ms/step - loss: 0.2263
Epoch 97/100
```

```
7/7 [==============================] - 5s 658ms/step - loss: 0.1650
Epoch 98/100
7/7 [==============================] - 5s 646ms/step - loss: 0.2181
Epoch 99/100
7/7 [==============================] - 5s 632ms/step - loss: 0.2300
Epoch 100/100
7/7 [==============================] - 4s 575ms/step - loss: 0.0724
Train accuracy: 100.00
Train accuracy: 74.00
```

## Testando redes neurais e *softmax*

Para classificação de múltiplas classes, tem-se um neurônio de saída para cada classe (como ilustrado no exemplo da Figura 1) e deve-se usar a operação Softmax antes de se calcular o custo (entropia cruzada ou cross-entropy como no exemplo anterior). Consute o capítulo 3.6 do livro para entender melhor. No caso de se usar softmax, deve-se usar a função **one_hot** para transformar a saída em logits.

A função **one_hot** transforma um escalar em um **hot encoder**, de acordo com o número de classes.

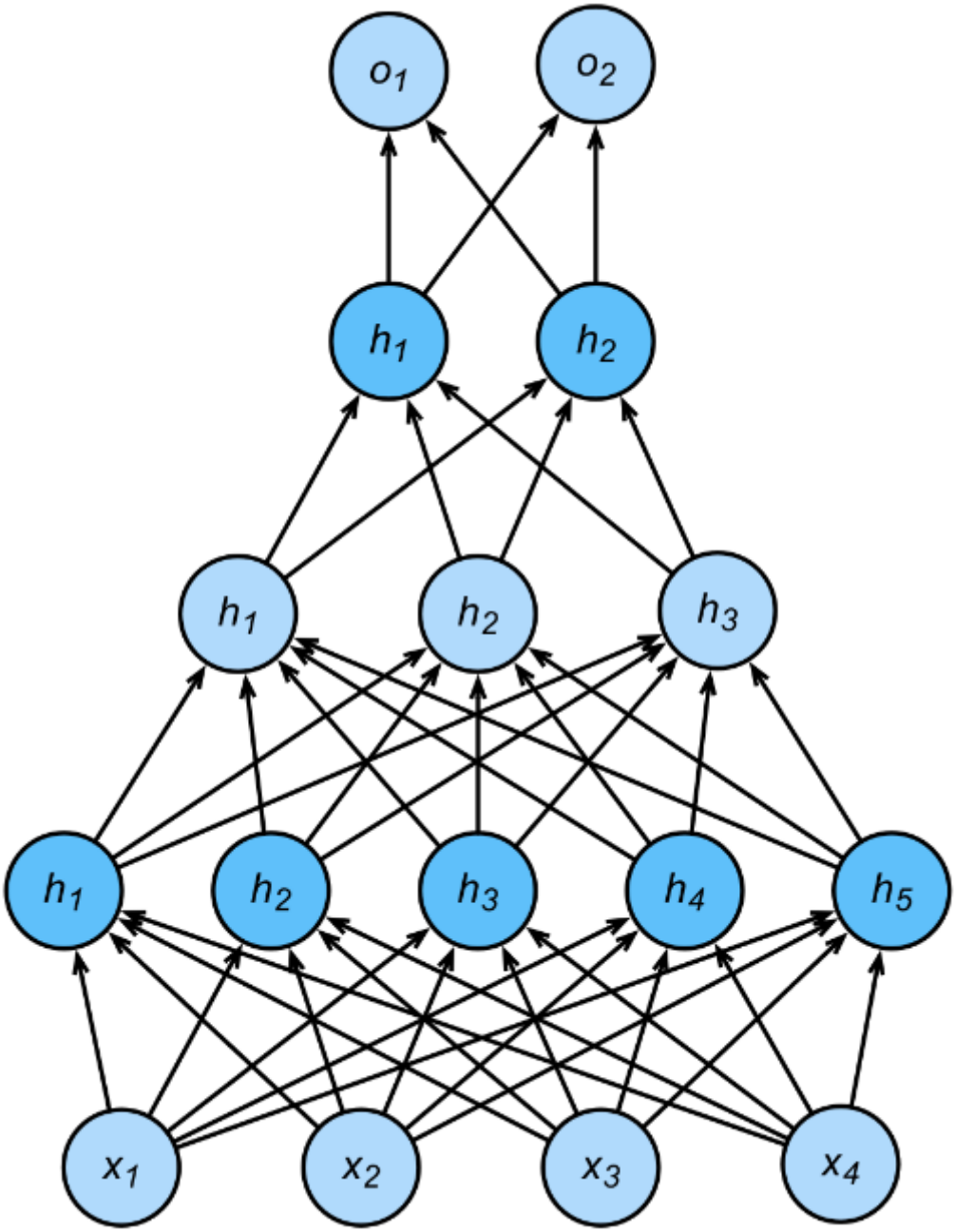



Figura 1: Rede neural dois neurônios de saída.

## Função de **one-hot encoded**

```
indices = [0, 1, 2]
depth = 3
tf.one_hot(indices, depth)  # output: [3 x 3]
```

```
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]], dtype=float32)>
```

### Saída esperada

```
[[1., 0., 0.],
 [0., 1., 0.],
 [0., 0., 1.]]
```

## **ToDo**: Função one-hot encoded (5pt)

O que a função one-hot encoded faz com o vetor na prática?

Tranforma cada possível valor do vetor(rótulo) em um vetor do tamanho da quantidade de valores(rótulos) possíveis. Cada vetor produzido possui em suas posições os valores de 0 ou 1, onde 1 representa que um rótulo está ativo. O vetor na realidade vira um vetor com esses vetores (uma matriz).

## Função **SoftMax**

A função softmax transforma a saída em uma distribuição de probabilidades. Assim, a soma de todas as saídas dos neurônio da última camada sempre vai ser igual a 1:

$$softmax(\mathbf{x}) = \frac{1}{\sum_{i=1}^{n} e^{x_i}} \cdot \begin{bmatrix} e^{x_1} \\ e^{x_2} \\ \vdots \\ e^{x_n} \end{bmatrix}$$

o gradiente para o custo usando-se a função softmax é trivial de se calcular:

$$dw = softmax(\mathbf{y_{pred}}) - y$$

```
indices = [-1., 0., 1.]

print(tf.nn.softmax(indices))
```

```
tf.Tensor([0.09003057 0.24472848 0.66524094], shape=(3,), dtype=float32)
```

**Saída esperada**

```
[0.09003057, 0.24472848, 0.66524094]
```

Perceba que esse código também funciona se você passar um lote (batch) de amostras

```
# Veja a saída abaixo
X = np.array([[10., 2., -3.],
              [-1., 5., -20.]])
print(tf.nn.softmax(X))
```

```
tf.Tensor(
[[9.99662391e-01 3.35349373e-04 2.25956630e-06]
 [2.47262316e-03 9.97527377e-01 1.38536042e-11]], shape=(2, 3), dtype=float64)
```

## **ToDo**: Função softmax (5pt)

O que a função softmax faz com o vetor na prática?

A função softmax transforma a saída em uma distribuição de probabilidades que somadas resulta em 1. Ela retorna um vetor como resultado, com a probabilidade(distribuição) de cada elemento do vetor passado como parâmetro. Quanto maior o elemento maior será sua probabilidade. Isso ajudará na classificação de múltiplas classes.

## Função de erro

Em seguida, deve-se computar o erro entre um vetor predito (**Y_pred**) e o vetor de real de rótulos (**Y_true**). para tal, deve-se usar cross entropy loss, ou verossimilhança negativa (negative log likelihood). A função **cross_entropy()** implementa a verossimilhança negativa.

```
tf.keras.losses.CategoricalCrossentropy()
```

Erro de uma predição bem ruim

```
y_true = [[1, 0, 0]]
y_pred = [[0.12, 4, 10]]

cce = tf.keras.losses.CategoricalCrossentropy()
print(cce(y_true, y_pred).numpy())
```

```
4.7678556
```

Erro de uma boa predição

```
y_true = [[1, 0, 0]]
y_pred = [[0.97, 0.01, 0.02]]
```

```
cce = tf.keras.losses.CategoricalCrossentropy()
print(cce(y_true, y_pred).numpy())
```

```
0.030459179
```

A função de erro também funciona para um lote de dados.

```
y_true = np.array([[0, 1, 0],
                   [1, 0, 0],
                   [0, 0, 1]])

y_pred = np.array([[0,   1,    0],
                   [.99, 0.01, 0],
                   [0,   0,    1]])

cce = tf.keras.losses.CategoricalCrossentropy()
print(cce(y_true, y_pred).numpy())
```

```
0.0033501784782856703
```

## **ToDo:** Função de erro (5pt)

Explique a função de erro *Categorical Cross-entropy*.

Ela calcula a perda de entropia cruzada entre os rótulos e as previsões, ou seja, é a medida da diferença entre duas distribuições, no caso a real e a predita. A função de perda requer as seguintes entradas: y_true(rótulo verdadeiro): isso é 0 ou 1, y_pred(valor previsto): previsão do modelo, ou seja, um único valor de ponto flutuante.

## Testando uma rede com uma camada oculta

Para esta atividade você deve usar uma loss (ou função de perda) baseada em softmax.

```
## Setando a seed
np.random.seed(1)
random.seed(1)

### Executar uma rede de 1 camada oculta ###
# Camadas da rede = [12288, 200, 1]

# Definição do modelo
model = Sequential()
```

```
model.add(Dense(200, input_shape=(12288,), activation='relu', name='CamadaOculta'))
model.add(Dense(2, activation='softmax', name='CamadaClassificacao'))

# Compilando o modelo
model.compile(loss='categorical_crossentropy', optimizer='adam')

# Imprimindo a arquitetura da rede proposta
model.summary()

# Treinando o modelo
model.fit(treino_x, tf.one_hot(treino_y.reshape(-1), 2), epochs=100)
```

```
Epoch 49/100
7/7 [==============================] - 0s 20ms/step - loss: 1.1084
Epoch 50/100
7/7 [==============================] - 0s 19ms/step - loss: 4.1968
Epoch 51/100
7/7 [==============================] - 0s 20ms/step - loss: 4.1626
Epoch 52/100
7/7 [==============================] - 0s 17ms/step - loss: 1.9273
Epoch 53/100
7/7 [==============================] - 0s 17ms/step - loss: 0.8222
Epoch 54/100
7/7 [==============================] - 0s 16ms/step - loss: 0.1440
Epoch 55/100
7/7 [==============================] - 0s 18ms/step - loss: 1.5554
Epoch 56/100
7/7 [==============================] - 0s 17ms/step - loss: 3.3536
Epoch 57/100
7/7 [==============================] - 0s 17ms/step - loss: 0.6949
Epoch 58/100
7/7 [==============================] - 0s 18ms/step - loss: 0.0328
Epoch 59/100
7/7 [==============================] - 0s 20ms/step - loss: 0.3164
Epoch 60/100
7/7 [==============================] - 0s 17ms/step - loss: 0.3408
Epoch 61/100
7/7 [==============================] - 0s 17ms/step - loss: 3.7032
Epoch 62/100
7/7 [==============================] - 0s 18ms/step - loss: 12.4676
Epoch 63/100
7/7 [==============================] - 0s 18ms/step - loss: 32.8054
Epoch 64/100
7/7 [==============================] - 0s 18ms/step - loss: 30.4529
Epoch 65/100
7/7 [==============================] - 0s 17ms/step - loss: 28.4256
Epoch 66/100
7/7 [==============================] - 0s 18ms/step - loss: 11.8998
Epoch 67/100
7/7 [==============================] - 0s 19ms/step - loss: 4.4435
Epoch 68/100
7/7 [==============================] - 0s 19ms/step - loss: 2.4682
Epoch 69/100
7/7 [==============================] - 0s 17ms/step - loss: 0.8535
Epoch 70/100
7/7 [==============================] - 0s 19ms/step - loss: 0.1792
Epoch 71/100
7/7 [==============================] - 0s 18ms/step - loss: 0.1061
Epoch 72/100
7/7 [==============================] - 0s 19ms/step - loss: 0.4757
```

```
Epoch 73/100
7/7 [==============================] - 0s 18ms/step - loss: 0.6915
Epoch 74/100
7/7 [==============================] - 0s 20ms/step - loss: 0.3287
Epoch 75/100
7/7 [==============================] - 0s 17ms/step - loss: 0.6772
Epoch 76/100
7/7 [==============================] - 0s 18ms/step - loss: 0.5357
Epoch 77/100
7/7 [==============================] - 0s 18ms/step - loss: 0.3576
Epoch 78/100
```

**Use os parâmetros treinados** para classificar as imagens de treinamento e teste e verificar a acurácia.

```
## Predição da rede
print(f'Acurácia no treino: {accuracy_score(treino_y.reshape(-1), np.argmax(model.predict(
print(f'Acurácia no teste: {accuracy_score(teste_y.reshape(-1), np.argmax(model.predict(te
```

```
Acurácia no treino: 100.00
Acurácia no teste: 70.00
```

**Resultado esperado:**

```
Acurácia treino = 100%
Acurácia teste = 70%
```

## **ToDo:** Testando outras redes (20pt)

Primero, implemente a rede de três camadas ocultas (mesma arquitetura utilizada com sigmoid). Por fim, repita o teste com uma arquitetura projetada por você, de preferência, bem profunda e mais larga. Plote a curva de custo (epochs vs loss) para cada um dos dois casos. O que você conclui?

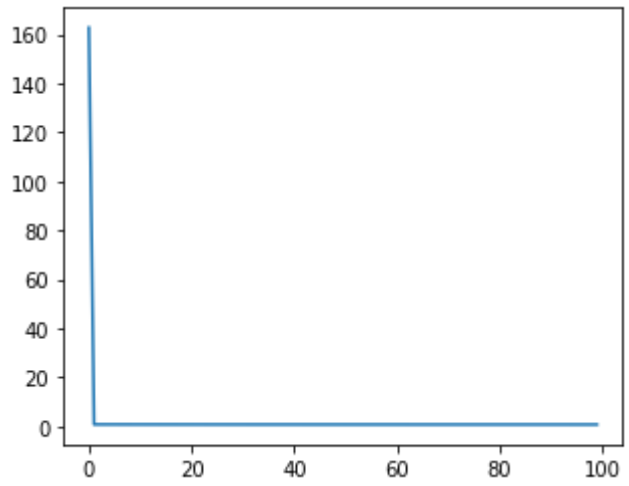
```
### Início do código ###
## Setando a seed
## Setando a seed
np.random.seed(1)
random.seed(1)

### Executar uma rede de 3 camadas ocultas ###
# Camadas da rede = [12288, 200, 70, 5, 2]

# Definição do modelo
model = Sequential()
model.add(Dense(200, input_shape=(12288,), activation='relu', name='CamadaOculta1'))
model.add(Dense(70, activation='relu', name='CamadaOculta2'))
model.add(Dense(5, activation='relu', name='CamadaOculta3'))
```

```
model.add(Dense(2, activation='softmax', name='CamadaClassificacao'))

# Compilando o modelo
model.compile(loss='categorical_crossentropy', optimizer='adam')

# Imprimindo a arquitetura da rede proposta
model.summary()

# Treinando o modelo
history = model.fit(treino_x, tf.one_hot(treino_y.reshape(-1), 2), epochs=100)
### Fim do código ###

## Predição da rede
print(f'Acurácia no treino: {accuracy_score(treino_y.reshape(-1), np.argmax(model.predict(
print(f'Acurácia no teste: {accuracy_score(teste_y.reshape(-1), np.argmax(model.predict(te
```

```
Epoch 73/100
7/7 [==============================] - 0s 23ms/step - loss: 0.6463
Epoch 74/100
7/7 [==============================] - 0s 24ms/step - loss: 0.6462
Epoch 75/100
7/7 [==============================] - 0s 24ms/step - loss: 0.6461
Epoch 76/100
7/7 [==============================] - 0s 21ms/step - loss: 0.6460
Epoch 77/100
7/7 [==============================] - 0s 22ms/step - loss: 0.6459
Epoch 78/100
7/7 [==============================] - 0s 23ms/step - loss: 0.6459
Epoch 79/100
7/7 [==============================] - 0s 17ms/step - loss: 0.6457
Epoch 80/100
7/7 [==============================] - 0s 17ms/step - loss: 0.6456
Epoch 81/100
7/7 [==============================] - 0s 16ms/step - loss: 0.6455
Epoch 82/100
7/7 [==============================] - 0s 17ms/step - loss: 0.6454
Epoch 83/100
7/7 [==============================] - 0s 18ms/step - loss: 0.6453
Epoch 84/100
7/7 [==============================] - 0s 16ms/step - loss: 0.6453
Epoch 85/100
7/7 [==============================] - 0s 17ms/step - loss: 0.6452
Epoch 86/100
7/7 [==============================] - 0s 17ms/step - loss: 0.6451
Epoch 87/100
7/7 [==============================] - 0s 17ms/step - loss: 0.6451
Epoch 88/100
7/7 [==============================] - 0s 17ms/step - loss: 0.6451
Epoch 89/100
7/7 [==============================] - 0s 17ms/step - loss: 0.6449
Epoch 90/100
7/7 [==============================] - 0s 18ms/step - loss: 0.6449
Epoch 91/100
7/7 [==============================] - 0s 18ms/step - loss: 0.6448
Epoch 92/100
7/7 [==============================] - 0s 18ms/step - loss: 0.6448
Epoch 93/100
7/7 [==============================] - 0s 18ms/step - loss: 0.6447
Epoch 94/100
```

```
7/7 [==============================] - 0s 18ms/step - loss: 0.6448
Epoch 95/100
7/7 [==============================] - 0s 22ms/step - loss: 0.6446
Epoch 96/100
7/7 [==============================] - 0s 23ms/step - loss: 0.6446
Epoch 97/100
7/7 [==============================] - 0s 22ms/step - loss: 0.6446
Epoch 98/100
7/7 [==============================] - 0s 23ms/step - loss: 0.6445
Epoch 99/100
7/7 [==============================] - 0s 22ms/step - loss: 0.6445
Epoch 100/100
7/7 [==============================] - 0s 22ms/step - loss: 0.6445
Acurácia no treino: 65.55
Acurácia no teste: 34.00
```

```
plt.plot(history.epoch, history.history['loss'], label="Distribuição do Erro")
```




```
### Início do código ###
## Setando a seed
## Setando a seed
np.random.seed(1)
random.seed(1)

### Executar uma rede de 7 camadas ocultas ###
# Camadas da rede = [12288, 500, 200, 70, 50, 20, 10 5, 2]

# Definição do modelo
model = Sequential()
model.add(Dense(500, input_shape=(12288,), activation='relu', name='CamadaOculta1'))
model.add(Dense(200, activation='relu', name='CamadaOculta2'))
model.add(Dense(70, activation='relu', name='CamadaOculta3'))
model.add(Dense(50, activation='relu', name='CamadaOculta4'))
model.add(Dense(10, activation='relu', name='CamadaOculta5'))
model.add(Dense(5, activation='relu', name='CamadaOculta6'))
model.add(Dense(2, activation='softmax', name='CamadaClassificacao'))

# Compilando o modelo
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

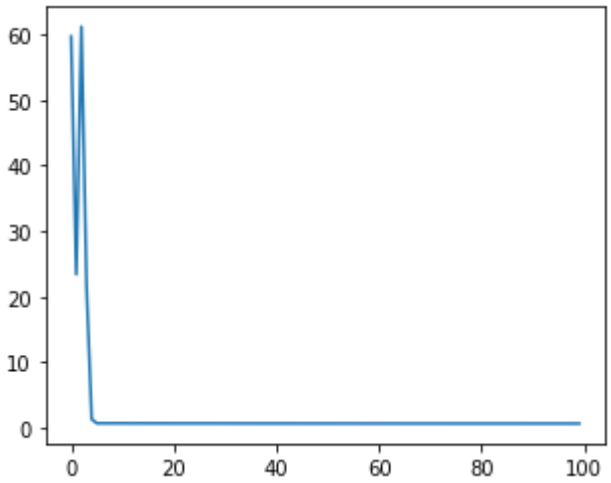
```
# Imprimindo a arquitetura da rede proposta
model.summary()

# Treinando o modelo
history = model.fit(treino_x, tf.one_hot(treino_y.reshape(-1), 2), epochs=100)
### Fim do código ###

## Predição da rede
print(f'Acurácia no treino: {accuracy_score(treino_y.reshape(-1), np.argmax(model.predict(
print(f'Acurácia no teste: {accuracy_score(teste_y.reshape(-1), np.argmax(model.predict(te
```

```
Epoch 73/100
7/7 [==============================] - 0s 42ms/step - loss: 0.6496
Epoch 74/100
7/7 [==============================] - 0s 45ms/step - loss: 0.6494
Epoch 75/100
7/7 [==============================] - 1s 123ms/step - loss: 0.6492
Epoch 76/100
7/7 [==============================] - 1s 136ms/step - loss: 0.6490
Epoch 77/100
7/7 [==============================] - 1s 83ms/step - loss: 0.6488
Epoch 78/100
7/7 [==============================] - 1s 86ms/step - loss: 0.6486
Epoch 79/100
7/7 [==============================] - 1s 91ms/step - loss: 0.6485
Epoch 80/100
7/7 [==============================] - 1s 77ms/step - loss: 0.6482
Epoch 81/100
7/7 [==============================] - 0s 66ms/step - loss: 0.6481
Epoch 82/100
7/7 [==============================] - 1s 78ms/step - loss: 0.6480
Epoch 83/100
7/7 [==============================] - 1s 83ms/step - loss: 0.6478
Epoch 84/100
7/7 [==============================] - 0s 48ms/step - loss: 0.6477
Epoch 85/100
7/7 [==============================] - 0s 40ms/step - loss: 0.6475
Epoch 86/100
7/7 [==============================] - 0s 40ms/step - loss: 0.6474
Epoch 87/100
7/7 [==============================] - 0s 41ms/step - loss: 0.6473
Epoch 88/100
7/7 [==============================] - 0s 58ms/step - loss: 0.6472
Epoch 89/100
7/7 [==============================] - 0s 56ms/step - loss: 0.6470
Epoch 90/100
7/7 [==============================] - 0s 56ms/step - loss: 0.6469
Epoch 91/100
7/7 [==============================] - 0s 52ms/step - loss: 0.6467
Epoch 92/100
7/7 [==============================] - 0s 56ms/step - loss: 0.6466
Epoch 93/100
7/7 [==============================] - 0s 57ms/step - loss: 0.6465
Epoch 94/100
7/7 [==============================] - 0s 54ms/step - loss: 0.6463
Epoch 95/100
7/7 [==============================] - 0s 60ms/step - loss: 0.6462
Epoch 96/100
7/7 [==============================] - 0s 57ms/step - loss: 0.6462
Epoch 97/100
```

```
7/7 [==============================] - 0s 57ms/step - loss: 0.6460
Epoch 98/100
7/7 [==============================] - 0s 55ms/step - loss: 0.6459
Epoch 99/100
7/7 [==============================] - 0s 53ms/step - loss: 0.6458
Epoch 100/100
7/7 [==============================] - 0s 58ms/step - loss: 0.6458
Acurácia no treino: 65.55
Acurácia no teste: 34.00
```

```
plt.plot(history.epoch, history.history['loss'], label="Distribuição do Erro")
```

```
[<matplotlib.lines.Line2D at 0x7f29c09257d0>]
```




É possível concluir que em redes neurais mais largas e profundas o erro no início das épocas sofre mais variação para depois convergir a um determinado erro menor.

## **ToDo:** Variando alguns hiperparâmetros (20pt)

Usando o framework do tensorflow/keras, altere os hiperparâmetros e veja o impacto (gere pelo menos dois novos modelos):

- learning rate,
- Algoritmo de otimização (SGD com momento, ADAM, ADADELTA, RMSPROP),
- inicialização dos pesos: inicialiação aleatória vs uniforme,
- Funções de ativação : troque a sigmoid por (ReLU, GELU, Leaky RELU),

```
### Início do código ###
## Setando a seed
np.random.seed(1)
random.seed(1)

### Executar uma rede de 1 camada oculta ###
# Camadas da rede = [12288, 200, 1]

# Definição do modelo
```

```
model = Sequential()
model.add(Dense(200, input_shape=(12288,), activation='relu', name='CamadaOculta', kernel_
model.add(Dense(1, activation='sigmoid', name='CamadaClassificacao'))

# Compilando o modelo
model.compile(loss='binary_crossentropy', optimizer='sgd')

# Imprimindo a arquitetura da rede proposta
model.summary()

# Treinando o modelo
model.fit(treino_x, treino_y.reshape(-1), epochs=100)
### Fim do código ###

## Predição da rede
print(f'Acurácia no treino: {accuracy_score(treino_y.reshape(-1), np.round(model.predict(t
print(f'Acurácia no teste: {accuracy_score(teste_y.reshape(-1), np.round(model.predict(tes
```

```
7/7 [==============================] - 0s 22ms/step - loss: 0.6258
Epoch 35/100
7/7 [==============================] - 0s 24ms/step - loss: 0.6254
Epoch 36/100
7/7 [==============================] - 0s 21ms/step - loss: 0.6251
Epoch 37/100
7/7 [==============================] - 0s 21ms/step - loss: 0.6248
Epoch 38/100
7/7 [==============================] - 0s 19ms/step - loss: 0.6245
Epoch 39/100
7/7 [==============================] - 0s 20ms/step - loss: 0.6242
Epoch 40/100
7/7 [==============================] - 0s 23ms/step - loss: 0.6239
Epoch 41/100
7/7 [==============================] - 0s 17ms/step - loss: 0.6236
Epoch 42/100
7/7 [==============================] - 0s 24ms/step - loss: 0.6233
Epoch 43/100
7/7 [==============================] - 0s 21ms/step - loss: 0.6230
Epoch 44/100
7/7 [==============================] - 0s 17ms/step - loss: 0.6228
Epoch 45/100
7/7 [==============================] - 0s 17ms/step - loss: 0.6225
Epoch 46/100
7/7 [==============================] - 0s 22ms/step - loss: 0.6223
Epoch 47/100
7/7 [==============================] - 0s 21ms/step - loss: 0.6220
Epoch 48/100
7/7 [==============================] - 0s 16ms/step - loss: 0.6218
Epoch 49/100
7/7 [==============================] - 0s 22ms/step - loss: 0.6216
Epoch 50/100
7/7 [==============================] - 0s 22ms/step - loss: 0.6214
Epoch 51/100
7/7 [==============================] - 0s 22ms/step - loss: 0.6211
Epoch 52/100
7/7 [==============================] - 0s 21ms/step - loss: 0.6209
Epoch 53/100
7/7 [==============================] - 0s 23ms/step - loss: 0.6207
Epoch 54/100
7/7 [==============================] - 0s 29ms/step - loss: 0.6205
```

```
Epoch 55/100
7/7 [==============================] - 0s 29ms/step - loss: 0.6204
Epoch 56/100
7/7 [==============================] - 0s 17ms/step - loss: 0.6202
Epoch 57/100
7/7 [==============================] - 0s 16ms/step - loss: 0.6200
Epoch 58/100
7/7 [==============================] - 0s 20ms/step - loss: 0.6199
Epoch 59/100
7/7 [==============================] - 0s 21ms/step - loss: 0.6197
Epoch 60/100
7/7 [==============================] - 0s 15ms/step - loss: 0.6195
Epoch 61/100
7/7 [==============================] - 0s 13ms/step - loss: 0.6194
Epoch 62/100
7/7 [==============================] - 0s 14ms/step - loss: 0.6192
Epoch 63/100
```

```
### Início do código ###
## Setando a seed
np.random.seed(1)
random.seed(1)

### Executar uma rede de 1 camada oculta ###
# Camadas da rede = [12288, 200, 1]

# Definição do modelo
model = Sequential()
model.add(Dense(200, input_shape=(12288,), activation='relu', name='CamadaOculta', kernel_
model.add(Dense(1, activation='sigmoid', name='CamadaClassificacao'))

# Compilando o modelo
model.compile(loss='binary_crossentropy', optimizer='adadelta')

# Imprimindo a arquitetura da rede proposta
model.summary()

# Treinando o modelo
model.fit(treino_x, treino_y.reshape(-1), epochs=100)
### Fim do código ###

## Predição da rede
print(f'Acurácia no treino: {accuracy_score(treino_y.reshape(-1), np.round(model.predict(t
print(f'Acurácia no teste: {accuracy_score(teste_y.reshape(-1), np.round(model.predict(tes
```

```
Epoch 73/100
7/7 [==============================] - 0s 21ms/step - loss: 3151.1428
Epoch 74/100
7/7 [==============================] - 0s 19ms/step - loss: 3130.5889
Epoch 75/100
7/7 [==============================] - 0s 19ms/step - loss: 3108.9844
Epoch 76/100
7/7 [==============================] - 0s 19ms/step - loss: 3085.6809
Epoch 77/100
7/7 [==============================] - 0s 19ms/step - loss: 3064.0159
Epoch 78/100
7/7 [==============================] - 0s 19ms/step - loss: 3043.0417
Epoch 79/100
```

```
Epoch 79/100
7/7 [==============================] - 0s 19ms/step - loss: 3024.2070
Epoch 80/100
7/7 [==============================] - 0s 22ms/step - loss: 3005.1292
Epoch 81/100
7/7 [==============================] - 0s 19ms/step - loss: 2986.8103
Epoch 82/100
7/7 [==============================] - 0s 19ms/step - loss: 2966.0745
Epoch 83/100
7/7 [==============================] - 0s 19ms/step - loss: 2945.2412
Epoch 84/100
7/7 [==============================] - 0s 20ms/step - loss: 2926.8008
Epoch 85/100
7/7 [==============================] - 0s 20ms/step - loss: 2908.7827
Epoch 86/100
7/7 [==============================] - 0s 19ms/step - loss: 2891.3220
Epoch 87/100
7/7 [==============================] - 0s 21ms/step - loss: 2873.6104
Epoch 88/100
7/7 [==============================] - 0s 19ms/step - loss: 2857.2451
Epoch 89/100
7/7 [==============================] - 0s 26ms/step - loss: 2841.6704
Epoch 90/100
7/7 [==============================] - 0s 24ms/step - loss: 2823.5933
Epoch 91/100
7/7 [==============================] - 0s 23ms/step - loss: 2806.6956
Epoch 92/100
7/7 [==============================] - 0s 25ms/step - loss: 2790.2512
Epoch 93/100
7/7 [==============================] - 0s 26ms/step - loss: 2774.4099
Epoch 94/100
7/7 [==============================] - 0s 23ms/step - loss: 2758.3894
Epoch 95/100
7/7 [==============================] - 0s 25ms/step - loss: 2742.2302
Epoch 96/100
7/7 [==============================] - 0s 23ms/step - loss: 2727.5994
Epoch 97/100
7/7 [==============================] - 0s 25ms/step - loss: 2713.4397
Epoch 98/100
7/7 [==============================] - 0s 25ms/step - loss: 2698.2212
Epoch 99/100
7/7 [==============================] - 0s 27ms/step - loss: 2684.6353
Epoch 100/100
7/7 [==============================] - 0s 25ms/step - loss: 2671.1672
Acurácia no treino: 58.37
Acurácia no teste: 34.00
```

```
### Início do código ###
## Setando a seed
np.random.seed(1)
random.seed(1)

### Executar uma rede de 1 camada oculta ###
# Camadas da rede = [12288, 200, 1]

# Definição do modelo
model = Sequential()
model.add(Dense(200, input_shape=(12288,), activation='relu', name='CamadaOculta', kernel_
```

```
model.add(Dense(1, activation='sigmoid', name='CamadaClassificacao'))

# Compilando o modelo
model.compile(loss='binary_crossentropy', optimizer='RMSPROP')

# Imprimindo a arquitetura da rede proposta
model.summary()

# Treinando o modelo
model.fit(treino_x, treino_y.reshape(-1), epochs=100)
### Fim do código ###

## Predição da rede
print(f'Acurácia no treino: {accuracy_score(treino_y.reshape(-1), np.round(model.predict(t
print(f'Acurácia no teste: {accuracy_score(teste_y.reshape(-1), np.round(model.predict(tes
```

```
Epoch 73/100
7/7 [==============================] - 0s 26ms/step - loss: 0.0000e+00
Epoch 74/100
7/7 [==============================] - 0s 29ms/step - loss: 0.0000e+00
Epoch 75/100
7/7 [==============================] - 0s 27ms/step - loss: 0.0000e+00
Epoch 76/100
7/7 [==============================] - 0s 26ms/step - loss: 0.0000e+00
Epoch 77/100
7/7 [==============================] - 0s 26ms/step - loss: 0.0000e+00
Epoch 78/100
7/7 [==============================] - 0s 36ms/step - loss: 0.0000e+00
Epoch 79/100
7/7 [==============================] - 0s 33ms/step - loss: 0.0000e+00
Epoch 80/100
7/7 [==============================] - 0s 35ms/step - loss: 0.0000e+00
Epoch 81/100
7/7 [==============================] - 0s 34ms/step - loss: 0.0000e+00
Epoch 82/100
7/7 [==============================] - 0s 35ms/step - loss: 0.0000e+00
Epoch 83/100
7/7 [==============================] - 0s 35ms/step - loss: 0.0000e+00
Epoch 84/100
7/7 [==============================] - 0s 34ms/step - loss: 0.0000e+00
Epoch 85/100
7/7 [==============================] - 0s 34ms/step - loss: 0.0000e+00
Epoch 86/100
7/7 [==============================] - 0s 36ms/step - loss: 0.0000e+00
Epoch 87/100
7/7 [==============================] - 0s 36ms/step - loss: 0.0000e+00
Epoch 88/100
7/7 [==============================] - 0s 35ms/step - loss: 0.0000e+00
Epoch 89/100
7/7 [==============================] - 0s 36ms/step - loss: 0.0000e+00
Epoch 90/100
7/7 [==============================] - 0s 30ms/step - loss: 0.0000e+00
Epoch 91/100
7/7 [==============================] - 0s 29ms/step - loss: 0.0000e+00
Epoch 92/100
7/7 [==============================] - 0s 28ms/step - loss: 0.0000e+00
Epoch 93/100
7/7 [==============================] - 0s 28ms/step - loss: 0.0000e+00
Epoch 94/100
```

```
7/7 [==============================] - 0s 28ms/step - loss: 0.0000e+00
Epoch 95/100
7/7 [==============================] - 0s 26ms/step - loss: 0.0000e+00
Epoch 96/100
7/7 [==============================] - 0s 29ms/step - loss: 0.0000e+00
Epoch 97/100
7/7 [==============================] - 0s 26ms/step - loss: 0.0000e+00
Epoch 98/100
7/7 [==============================] - 0s 27ms/step - loss: 0.0000e+00
Epoch 99/100
7/7 [==============================] - 0s 27ms/step - loss: 0.0000e+00
Epoch 100/100
7/7 [==============================] - 0s 27ms/step - loss: 0.0000e+00
Acurácia no treino: 100.00
Acurácia no teste: 66.00
```

```
### Início do código ###
## Setando a seed
np.random.seed(1)
random.seed(1)

### Executar uma rede de 1 camada oculta ###
# Camadas da rede = [12288, 200, 1]

# Definição do modelo
model = Sequential()
model.add(Dense(200, input_shape=(12288,), activation='gelu', name='CamadaOculta', kernel_
model.add(Dense(1, activation='sigmoid', name='CamadaClassificacao'))

# Compilando o modelo
model.compile(loss='binary_crossentropy', optimizer='RMSPROP')

# Imprimindo a arquitetura da rede proposta
model.summary()

# Treinando o modelo
model.fit(treino_x, treino_y.reshape(-1), epochs=100)
### Fim do código ###

## Predição da rede
print(f'Acurácia no treino: {accuracy_score(treino_y.reshape(-1), np.round(model.predict(t
print(f'Acurácia no teste: {accuracy_score(teste_y.reshape(-1), np.round(model.predict(tes
```

```
Epoch 73/100
7/7 [==============================] - 0s 30ms/step - loss: 0.0000e+00
Epoch 74/100
7/7 [==============================] - 0s 28ms/step - loss: 0.0000e+00
Epoch 75/100
7/7 [==============================] - 0s 27ms/step - loss: 0.0000e+00
Epoch 76/100
7/7 [==============================] - 0s 28ms/step - loss: 0.0000e+00
Epoch 77/100
7/7 [==============================] - 0s 27ms/step - loss: 0.0000e+00
Epoch 78/100
7/7 [==============================] - 0s 29ms/step - loss: 0.0000e+00
Epoch 79/100
7/7 [==============================] - 0s 27ms/step - loss: 0.0000e+00
Epoch 80/100
```

```
7/7 [==============================] - 0s 27ms/step - loss: 0.0000e+00
Epoch 81/100
7/7 [==============================] - 0s 28ms/step - loss: 0.0000e+00
Epoch 82/100
7/7 [==============================] - 0s 39ms/step - loss: 0.0000e+00
Epoch 83/100
7/7 [==============================] - 0s 37ms/step - loss: 0.0000e+00
Epoch 84/100
7/7 [==============================] - 0s 35ms/step - loss: 0.0000e+00
Epoch 85/100
7/7 [==============================] - 0s 36ms/step - loss: 0.0000e+00
Epoch 86/100
7/7 [==============================] - 0s 28ms/step - loss: 0.0000e+00
Epoch 87/100
7/7 [==============================] - 0s 29ms/step - loss: 0.0000e+00
Epoch 88/100
7/7 [==============================] - 0s 26ms/step - loss: 0.0000e+00
Epoch 89/100
7/7 [==============================] - 0s 27ms/step - loss: 0.0000e+00
Epoch 90/100
7/7 [==============================] - 0s 27ms/step - loss: 0.0000e+00
Epoch 91/100
7/7 [==============================] - 0s 29ms/step - loss: 0.0000e+00
Epoch 92/100
7/7 [==============================] - 0s 29ms/step - loss: 0.0000e+00
Epoch 93/100
7/7 [==============================] - 0s 28ms/step - loss: 0.0000e+00
Epoch 94/100
7/7 [==============================] - 0s 28ms/step - loss: 0.0000e+00
Epoch 95/100
7/7 [==============================] - 0s 28ms/step - loss: 0.0000e+00
Epoch 96/100
7/7 [==============================] - 0s 36ms/step - loss: 0.0000e+00
Epoch 97/100
7/7 [==============================] - 0s 36ms/step - loss: 0.0000e+00
Epoch 98/100
7/7 [==============================] - 0s 35ms/step - loss: 0.0000e+00
Epoch 99/100
7/7 [==============================] - 0s 36ms/step - loss: 0.0000e+00
Epoch 100/100
7/7 [==============================] - 0s 36ms/step - loss: 0.0000e+00
Acurácia no treino: 100.00
Acurácia no teste: 66.00
```

```
### Início do código ###
## Setando a seed
np.random.seed(1)
random.seed(1)

### Executar uma rede de 1 camada oculta ###
# Camadas da rede = [12288, 200, 1]

# Definição do modelo
model = Sequential()
model.add(Dense(200, input_shape=(12288,), activation='gelu', name='CamadaOculta', kernel_
model.add(Dense(1, activation='sigmoid', name='CamadaClassificacao'))
```

```
# Compilando o modelo
model.compile(loss='binary_crossentropy', optimizer='adam')

# Imprimindo a arquitetura da rede proposta
model.summary()

# Treinando o modelo
model.fit(treino_x, treino_y.reshape(-1), epochs=100)
### Fim do código ###

## Predição da rede
print(f'Acurácia no treino: {accuracy_score(treino_y.reshape(-1), np.round(model.predict(t
print(f'Acurácia no teste: {accuracy_score(teste_y.reshape(-1), np.round(model.predict(tes
```

```
Epoch 73/100
7/7 [==============================] - 0s 22ms/step - loss: 0.0000e+00
Epoch 74/100
7/7 [==============================] - 0s 22ms/step - loss: 0.0000e+00
Epoch 75/100
7/7 [==============================] - 0s 22ms/step - loss: 0.0000e+00
Epoch 76/100
7/7 [==============================] - 0s 24ms/step - loss: 0.0000e+00
Epoch 77/100
7/7 [==============================] - 0s 23ms/step - loss: 0.0000e+00
Epoch 78/100
7/7 [==============================] - 0s 18ms/step - loss: 0.0000e+00
Epoch 79/100
7/7 [==============================] - 0s 20ms/step - loss: 0.0000e+00
Epoch 80/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0000e+00
Epoch 81/100
7/7 [==============================] - 0s 18ms/step - loss: 0.0000e+00
Epoch 82/100
7/7 [==============================] - 0s 17ms/step - loss: 0.0000e+00
Epoch 83/100
7/7 [==============================] - 0s 18ms/step - loss: 0.0000e+00
Epoch 84/100
7/7 [==============================] - 0s 18ms/step - loss: 0.0000e+00
Epoch 85/100
7/7 [==============================] - 0s 19ms/step - loss: 0.0000e+00
Epoch 86/100
7/7 [==============================] - 0s 25ms/step - loss: 0.0000e+00
Epoch 87/100
7/7 [==============================] - 0s 22ms/step - loss: 0.0000e+00
Epoch 88/100
7/7 [==============================] - 0s 22ms/step - loss: 0.0000e+00
Epoch 89/100
7/7 [==============================] - 0s 22ms/step - loss: 0.0000e+00
Epoch 90/100
7/7 [==============================] - 0s 21ms/step - loss: 0.0000e+00
Epoch 91/100
7/7 [==============================] - 0s 22ms/step - loss: 0.0000e+00
Epoch 92/100
7/7 [==============================] - 0s 19ms/step - loss: 0.0000e+00
Epoch 93/100
7/7 [==============================] - 0s 20ms/step - loss: 0.0000e+00
Epoch 94/100
```

```
7/7 [==============================] - 0s 18ms/step - loss: 0.0000e+00
Epoch 95/100
7/7 [==============================] - 0s 16ms/step - loss: 0.0000e+00
Epoch 96/100
7/7 [==============================] - 0s 17ms/step - loss: 0.0000e+00
Epoch 97/100
7/7 [==============================] - 0s 18ms/step - loss: 0.0000e+00
Epoch 98/100
7/7 [==============================] - 0s 19ms/step - loss: 0.0000e+00
Epoch 99/100
7/7 [==============================] - 0s 19ms/step - loss: 0.0000e+00
Epoch 100/100
7/7 [==============================] - 0s 24ms/step - loss: 0.0000e+00
Acurácia no treino: 100.00
Acurácia no teste: 56.00
```

```
###·Início·do·código·###
##·Setando·a·seed
np.random.seed(1)
random.seed(1)

###·Executar·uma·rede·de·1·camada·oculta·###
#·Camadas·da·rede·=·[12288,·200,·1]·

#·Definição·do·modelo
model·=·Sequential()
model.add(Dense(200,·input_shape=(12288,),·activation='relu',·name='CamadaOculta'))
model.add(Dense(1,·activation='sigmoid',·name='CamadaClassificacao'))

#·Compilando·o·modelo
model.compile(loss='binary_crossentropy',·optimizer='adam')

#·Imprimindo·a·arquitetura·da·rede·proposta
model.summary()

#·Treinando·o·modelo
model.fit(treino_x,·treino_y.reshape(-1),·epochs=100)
###·Fim·do·código·###

##·Predição·da·rede
print(f'Acurácia·no·treino:·{accuracy_score(treino_y.reshape(-1),·np.round(model.predict(t
print(f'Acurácia·no·teste:·{accuracy_score(teste_y.reshape(-1),·np.round(model.predict(tes
```

```
Model: "sequential_43"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
CamadaOculta (Dense)         (None, 200)               2457800
_________________________________________________________________
CamadaClassificacao (Dense)  (None, 1)                 201
=================================================================
Total params: 2,458,001
Trainable params: 2,458,001
Non-trainable params: 0
```

```
_________________________________________________________________
Epoch 1/100
7/7 [==============================] - 1s 21ms/step - loss: 1899.5931
Epoch 2/100
7/7 [==============================] - 0s 21ms/step - loss: 609.6093
Epoch 3/100
7/7 [==============================] - 0s 28ms/step - loss: 267.2789
Epoch 4/100
7/7 [==============================] - 0s 26ms/step - loss: 152.4326
Epoch 5/100
7/7 [==============================] - 0s 24ms/step - loss: 110.7168
Epoch 6/100
7/7 [==============================] - 0s 27ms/step - loss: 36.1824
Epoch 7/100
7/7 [==============================] - 0s 24ms/step - loss: 46.9148
Epoch 8/100
7/7 [==============================] - 0s 25ms/step - loss: 60.9487
Epoch 9/100
7/7 [==============================] - 0s 32ms/step - loss: 33.9603
Epoch 10/100
7/7 [==============================] - 0s 23ms/step - loss: 22.7876
Epoch 11/100
7/7 [==============================] - 0s 25ms/step - loss: 13.9051
Epoch 12/100
7/7 [==============================] - 0s 23ms/step - loss: 17.7887
Epoch 13/100
7/7 [==============================] - 0s 23ms/step - loss: 27.3105
Epoch 14/100
7/7 [==============================] - 0s 26ms/step - loss: 19.7043
Epoch 15/100
7/7 [==============================] - 0s 21ms/step - loss: 12.6698
Epoch 16/100
7/7 [==============================] - 0s 22ms/step - loss: 17.2494
Epoch 17/100
7/7 [==============================] - 0s 23ms/step - loss: 13.4265
Epoch 18/100
7/7 [==============================] - 0s 22ms/step - loss: 18.8268
Epoch 19/100
7/7 [==============================] - 0s 21ms/step - loss: 4.0057
Epoch 20/100
7/7 [==============================] - 0s 18ms/step - loss: 2.8827
Epoch 21/100
7/7 [==============================] - 0s 20ms/step - loss: 2.0993
Epoch 22/100
7/7 [==============================] - 0s 17ms/step - loss: 1.2552
Epoch 23/100
```

## **ToDo:** Analisando redes treinadas (5pt)

Qual combinação rendeu o melhor resultado? Tente explicar o por que.

A melhor combinação para um rede de 1 camada oculta com 200 neurônios foi a rede que utilizou: inicialização de pesos uniformes, a função relu como função de ativação e o algoritmo de otimização adam. Isso acontece porque a função ReLU não ativa todos os neurônios ao mesmo tempo, isso significa que, ao mesmo tempo, apenas alguns neurônios

são ativados, tornando a rede esparsa e eficiente e fácil para a computação. Com a inicialização do pesos de maneira uniforme evita-se o problema com a saturação dos neurônios. Já o uso do Adam como otimizador garante eficiência ao trabalhar com problemas envolvendo muitos dados ou parâmetros, como neste caso.

✓ 15s conclusão: 21:30