

## ▼ Trabalho #3 - Treinamento customizado

Nesse trabalho você vai treinar uma RNA para prever se um tumor é maligno ou benigno usando o conjunto de dados "Breast Cancer Dataset", disponível no UCI Machine Learning Repository ([https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+\(original\)](https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original))).

Esse conjunto de dados foi obtido pelo Hospital da University de Wisconsin, Madison por: O. L. Mangasarian e W. H. Wolberg, "Cancer diagnosis via linear programming", SIAM News, Volume 23, Number 5, September 1990, pp 1 & 18.

Coloque o seu nome:

Nome: Gabriel Silvestre Mancini

### ▼ 1. Importar bibliotecas

Execute a célula abaixo para importar as principais bilbiotecas necessárias.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

### ▼ 2. Carregar dados

Esse conjunto de dados possui 699 exemplos, sendo que cada exemplo é composto por 10 características obtidas por exames de células detectados que podem ser da classe de cancer maligno ou benigno.

As características de cada exemplo são as seguintes:

1. Número de identificação da amostra: id
2. Espessura do aglomerado: 1 - 10
3. Uniformidade do tamanho da célula: 1 - 10
4. Uniformidade da forma celular: 1 - 10
5. Adesão Marginal: 1 - 10
6. Tamanho de célula epitelial única: 1 - 10
7. Núcleos expostos: 1 - 10
8. Cromatina Suave: 1 - 10
9. Nucléos normais: 1 - 10
10. Mitoses: 1 - 10
11. Classe: 2 para benigno e 4 para maligno

Execute a célula abaixo para carregar o conjunto de dados e criar um DataFrame Pandas. Para facilitar o entendimento dos dados vamos definir explicitamente os nomes das colunas porque o arquivo CSV original não possui o cabeçalho com os nomes das colunas.

```
DATASET_URL = "https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/breast-cancer-wisconsin.data"
data_file = tf.keras.utils.get_file("breast_cancer.csv", DATASET_URL)
col_names = ["id", "espessura", "un_tam_cel", "un_forma_cel", "adesao_marginal", "tam_cel_epit", "nucleos_exp", "cromatina", "nucleos_normais", "mitoses", "classe"]
df = pd.read_csv(data_file, names=col_names, header=None)
```

Execute as duas células abaixo para visualizar os dados e verificar o número de exemplos.

```
df.head()
```

	id	espessura	un_tam_cel	un_forma_cel	adesao_marginal	tam_cel_epit	nucleos_exp	cromatina	nucleos_normais	mitoses	classe
0	1000025	5	1	1	1	2	1	3	1	1	1
1	1002945	5	4	4	5	7	10	3	2	1	1
2	1015425	3	1	1	1	2	2	3	1	1	1
3	1016277	6	8	8	1	3	4	3	7	1	1
4	1017023	4	1	1	3	2	1	3	1	1	1

Próximas etapas: [Gerar código com df](#) [Ver gráficos recomendados](#) [New interactive sheet](#)

```
print('Dimensão dos dados:', df.shape)
```

Dimensão dos dados: (699, 11)

## ▼ 3. Pré-processamento dos dados

Os dados precisam ser pré-processados para poderem ser utilizados por uma RNA.

As seguintes etapas devem ser realizadas no processamento:

1. Retirar a coluna da identificação da amostra ("id");
2. Eliminar dados que possuem valores "desconhecidos";
3. Transformar classes de índices 2 e 4 para 0 e 1, 0 é benigno e 1 é maligno;
4. Dividir dados nos conjuntos de treinamento e teste;
5. Separar coluna das classes (dados de saídas reais) das outras colunas (dados de entradas);
6. Normalizar os dados de entrada;
7. Converter DataFrame Pandas para tf.Tensor.

### ▼ 3.1 Retirar coluna de identificação da amostra ("id")

Execute a célula baixo para realizar essa operação.

```
df.pop("id")
df.head()
```

Execution results

Próximas etapas: [Gerar código com df](#) [Ver gráficos recomendados](#) [New interactive sheet](#)

	espressura	un_tam_cel	un_forma_cel	adesao_marginal	tam_cel_epit	nucleos_exp	cromatina	nucleos_normais	mitoses	classe
0	5	1	1	1	2	1	3	1	1	2
1	5	4	4	5	7	10	3	2	1	2
2	3	1	1	1	2	2	3	1	1	2
3	6	8	8	1	3	4	3	7	1	2
4	4	1	1	3	2	1	3	1	1	2

### ▼ 3.2 Eliminar dados "desconhecidos"

Se você inspecionar os dados vai verificar que existem valores "desconhecidos" na coluna de "nucleos\_exp". Para verificar quais amostras possuem valores desconhecidos execute a célula abaixo.

```
df[df["nucleos_exp"] == '?']
```

	espressura	un_tam_cel	un_forma_cel	adesao_marginal	tam_cel_epit	nucleos_exp	cromatina	nucleos_normais	mitoses	classe
23	8	4	5	1	2	?	7	3	1	4
40	6	6	6	9	6	?	7	8	1	2
139	1	1	1	1	1	?	2	1	1	2
145	1	1	3	1	2	?	2	1	1	2
158	1	1	2	1	3	?	1	1	1	2
164	5	1	1	1	2	?	3	1	1	2
235	3	1	4	1	2	?	3	1	1	2
249	3	1	1	1	2	?	3	1	1	2
275	3	1	3	1	2	?	2	1	1	2
292	8	8	8	1	2	?	6	10	1	4
294	1	1	1	1	2	?	2	1	1	2
297	5	4	3	1	2	?	2	3	1	2
315	4	6	5	6	7	?	4	9	1	2
321	3	1	1	1	2	?	3	1	1	2
411	1	1	1	1	1	?	2	1	1	2
617	1	1	1	1	1	?	1	1	1	2

Deve-se eliminar as linhas que possuem dados desse tipo. Além disso, a coluna "nucleos\_exp" não é uma coluna numérica e, portanto, deve ser convertida para valores numéricos. Execute a célula abaixo para realizar essas operações.

```
# Elimina linhas com dados desconhecidos na coluna "nucleos_exp"
df = df[df["nucleos_exp"] != '?']
```

```
# Converte coluna "nucleo_exp" para valores numéricos
df.nucleos_exp = pd.to_numeric(df.nucleos_exp)
df
```

```
→ <ipython-input-7-1f0d4e6ea00e>:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

See the caveats in the documentation: [https://pandas.pydata.org/pandas-docs/stable/user\\_guide/indexing.html#returning-a-view-versus](https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus)

```
df.nucleos_exp = pd.to_numeric(df.nucleos_exp)
```

	espressura	un_tam_cel	un_forma_cel	adesao_marginal	tam_cel_epit	nucleos_exp	cromatina	nucleos_normais	mitoses	classe
0	5	1	1	1	2	1	3	1	1	2
1	5	4	4	5	7	10	3	2	1	2
2	3	1	1	1	2	2	3	1	1	2
3	6	8	8	1	3	4	3	7	1	2
4	4	1	1	3	2	1	3	1	1	2
...	...	...	...	...	...	...	...	...	...	...
694	3	1	1	1	3	2	1	1	1	2
695	2	1	1	1	2	1	1	1	1	2
696	5	10	10	3	7	3	8	10	2	4
697	4	8	6	4	3	4	10	6	1	4
698	4	8	8	5	4	5	10	4	1	4

683 rows × 10 columns

Próximas etapas: [Gerar código com df](#) [Ver gráficos recomendados](#) [New interactive sheet](#)

- Observe que o conjunto de dados agora tem 683 exemplos, ou seja, 16 exemplos foram retirados porque tinham valores "unknown".

### 3.3 Transformar código das classes de câncer

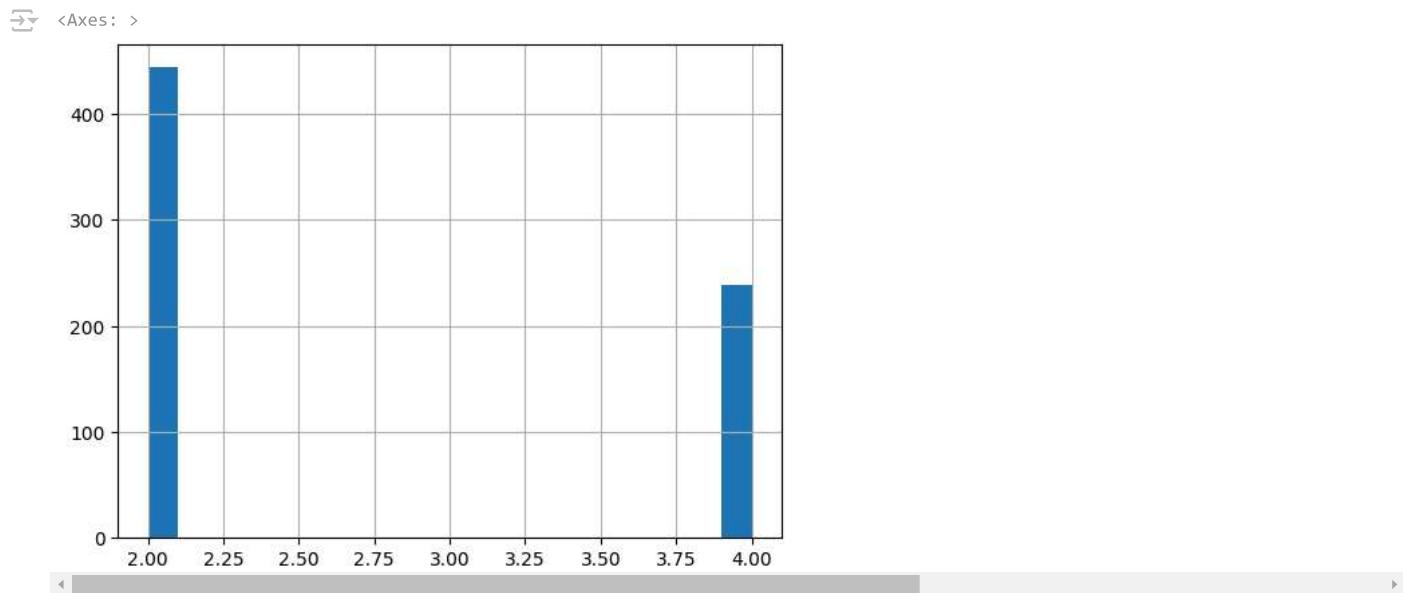
Primeiramente vamos verificar o número de exemplos de cada classe. Lembre que:

- Classe = 2 → câncer benigno
- Classe = 4 → câncer maligno

Observa-se que é importante fazer essa verificação porque se o número de exemplos das classes for muito desbalanceado temos que usar técnicas especiais para treinar a RNA, como já visto anteriormente.

Para visualizar o número de exemplos de cada classe vamos calcular e fazer o gráfico do histograma da coluna "classe".

```
df['classe'].hist(bins=20)
```



Para poder modelar esse problema como um problema de classificação binária, que detecta se o tumor é maligno ou não, temos que alterar os códigos das classes para o seguinte:

- Câncer benigno (2.0) = 0
- Câncer maligno (4.0) = 1

```
df['classe'] = np.where(df['classe'] == 2, 0, 1)
df
```

	espressura	un_tam_cel	un_forma_cel	adesao_marginal	tam_cel_epit	nucleos_exp	cromatina	nucleos_normais	mitoses	classe
0	5	1	1	1	2	1	3	1	1	0
1	5	4	4	5	7	10	3	2	1	0
2	3	1	1	1	2	2	3	1	1	0
3	6	8	8	1	3	4	3	7	1	0
4	4	1	1	3	2	1	3	1	1	0
...	...	...	...	...	...	...	...	...	...	...
694	3	1	1	1	3	2	1	1	1	0
695	2	1	1	1	2	1	1	1	1	0
696	5	10	10	3	7	3	8	10	2	1
697	4	8	6	4	3	4	10	6	1	1
698	4	8	8	5	4	5	10	4	1	1

683 rows × 10 columns

Próximas etapas: [Gerar código com df](#) [Ver gráficos recomendados](#) [New interactive sheet](#)

## Exercício #1: Dividir e embaralhar conjunto de dados

Vamos dividir o conjunto de dados em conjuntos de treinamento e teste. Como o número de amostras é pequeno, faremos a validação no conjunto de teste.

Nessa divisão vamos utilizar 80% dos dados como sendo de treinamento e 20% como sendo de teste/validação.

Para realizar essa divisão usaremos a função `train_test_split()` da biblioteca ScikitLearn. Observe que você deve usar essa função também para embaralhar aleatoriamente os dados.

```
# Para você fazer: Dividir e embaralhar dados
```

```
# Importa função para dividir conjunto de dados
from sklearn.model_selection import train_test_split

# Realiza divisão dos dados
# Inclua seu código aqui
train_data, test_data = train_test_split(df, test_size=0.2, random_state=666, shuffle=True)

# Verifica as dimensões completas dos conjuntos
print(f"Dimensão dos dados de treinamento: {train_data.shape}")
print(f"Dimensão dos dados de teste: {test_data.shape}")
```

→ Dimensão dos dados de treinamento: (546, 10)  
 Dimensão dos dados de teste: (137, 10)

### Saída esperada:

```
Dimensão dos dados de treinamento: (546, 10)
Dimensão dos dados de teste: (137, 10)
```

### 3.4 Separar coluna das classes (saída desejada)

Devemos separar a coluna das classes dos conjuntos de treinamento e teste para criar as saídas desejadas de treinamento e teste.

```
train_Y = train_data.pop("classe")
test_Y = test_data.pop("classe")
train_Y
```

	classe
203	0
450	0
319	0
388	0
314	0
...	...
460	0
429	0
72	0
444	0
243	0

546 rows × 1 columns

dtype: int64

Vamos calcular as estatísticas básicas das saídas dos conjuntos de treinamento e teste para verificar se ambos possuem a mesma distribuição.

```
print('Estatística das saídas de treinamento:\n', train_Y.describe())
print('\nEstatística das saídas de teste:\n', test_Y.describe())
```

→ Estatística das saídas de treinamento:

count	546.000000
mean	0.340659
std	0.474366
min	0.000000
25%	0.000000
50%	0.000000
75%	1.000000
max	1.000000
Name: classe, dtype:	float64

Estatística das saídas de teste:

count	137.000000
mean	0.386861
std	0.488819
min	0.000000
25%	0.000000
50%	0.000000
75%	1.000000

```
max      1.000000
Name: classe, dtype: float64
```

### ▼ 3.5 Normalizar dados de entrada

Antes de normalizar os dados de entrada é importante calcular as suas estatísticas básicas. Os valores de média e desvio padrão das características dos dados de entrada de treinamento serão usados para normalizar os dados de treinamento e de teste.

```
train_stats = train_data.describe()
train_stats = train_stats.transpose()
print(train_stats.shape)
train_stats
```

	count	mean	std	min	25%	50%	75%	max
espessura	546.0	4.406593	2.801617	1.0	2.0	4.0	6.0	10.0
un_tam_cel	546.0	3.109890	3.015375	1.0	1.0	1.0	4.0	10.0
un_forma_cel	546.0	3.184982	2.948586	1.0	1.0	1.0	5.0	10.0
adesao_marginal	546.0	2.783883	2.838302	1.0	1.0	1.0	3.0	10.0
tam_cel_epit	546.0	3.181319	2.193004	1.0	2.0	2.0	4.0	10.0
nucleos_exp	546.0	3.545788	3.649646	1.0	1.0	1.0	6.0	10.0
cromatina	546.0	3.382784	2.409080	1.0	2.0	3.0	4.0	10.0
nucleos_normais	546.0	2.787546	2.966899	1.0	1.0	1.0	3.0	10.0
mitoses	546.0	1.591575	1.700470	1.0	1.0	1.0	1.0	10.0

Próximas etapas: [Gerar código com train\\_stats](#) [Ver gráficos recomendados](#) [New interactive sheet](#)

### ▼ Exercícios #2: Normalizar dados de entrada

Os dados de entrada serão normalizados para terem média igual a zero e desvio padrão igual a 1. Assim, a normalização de cada coluna deve ser feita de acordo com a seguinte equação:

$$X_{norm,i} = \frac{(X_i - \mu_i)}{\sigma_i}$$

onde  $X_i$  é a  $i$ -ésima coluna dos dados,  $\mu_i$  é a média da  $i$ -ésima coluna,  $\sigma_i$  é o desvio padrão da  $i$ -ésima coluna e  $X_{norm,i}$  é a  $i$ -ésima coluna dos dados normalizada.

```
stats = train_stats.describe().transpose()
```

```
stats['mean']
```

	mean
count	546.000000
mean	3.108262
std	2.724775
min	1.000000
25%	1.333333
50%	1.666667
75%	4.000000
max	10.000000

```
dtype: float64
```

```
# Para você fazer: Normalizar dados
```

```
# Define função para normalizar as colunas
# Inclua seu código aqui
def normalizar_dados(df, stats):
    return (df - stats['mean']) / stats['std']
```

```
# Calcula estatísticas dos dados de treinamento
train_stats = train_data.describe().transpose()
```

```
# Normaliza dados de entrada de treinamento e teste
X_train_norm = normalizar_dados(train_data, train_stats)
X_test_norm = normalizar_dados(test_data, train_stats)

# Visualiza dados de entrada de treinamento normalizados
print("Dados de entrada de treinamento normalizados:")
print(X_train_norm.head())
print("")
print(X_train_norm.describe().transpose())

▼ Dados de entrada de treinamento normalizados:
  espessura un_tam_cel un_forma_cel adesao_marginal tam_cel_epit \
203  0.211809 -0.699711 -0.741027 -0.628503 -0.538676
450  -0.145128 -0.699711 -0.741027  0.076143 -0.994672
319  -0.145128  0.295190  0.276410  0.428466  1.285306
388  -0.859002 -0.699711 -0.741027 -0.628503 -0.538676
314  -1.215938 -0.699711 -0.741027 -0.628503 -0.994672

  nucleos_exp cromatina nucleos_normais mitoses
203   -0.697544 -0.158892      -0.602496 -0.347889
450   -0.697544 -0.573988      -0.602496 -0.347889
319    0.398453  1.501493     0.071608 -0.347889
388   -0.697544 -0.573988     -0.265444 -0.347889
314   -0.697544 -0.573988      -0.602496 -0.347889

  count      mean     std      min     25%     50% \
espessura  546.0 -1.057355e-16  1.0  -1.215938 -0.859002 -0.145128
un_tam_cel  546.0  2.440051e-17  1.0  -0.699711 -0.699711 -0.699711
un_forma_cel 546.0  8.336840e-17  1.0  -0.741027 -0.741027 -0.741027
adesao_marginal 546.0  4.229421e-17  1.0  -0.628503 -0.628503 -0.628503
tam_cel_epit   546.0  1.073622e-16  1.0  -0.994672 -0.538676 -0.538676
nucleos_exp    546.0  4.066751e-17  1.0  -0.697544 -0.697544 -0.697544
cromatina     546.0  -6.506802e-18  1.0  -0.989085 -0.573988 -0.158892
nucleos_normais 546.0  2.928061e-17  1.0  -0.602496 -0.602496 -0.602496
mitoses       546.0  5.856121e-17  1.0  -0.347889 -0.347889 -0.347889

  75%      max
espessura    0.568745  1.996492
un_tam_cel   0.295190  2.284992
un_forma_cel  0.615555  2.311284
adesao_marginal 0.076143  2.542406
tam_cel_epit   0.373315  3.109289
nucleos_exp    0.672452  1.768449
cromatina     0.256204  2.746781
nucleos_normais 0.071608  2.430974
mitoses       -0.347889  4.944765
```

### Saída esperada:

	count	mean	std	min	25%	50%	75%	max
espessura	546.0	-2.373966e-17	1.0	-1.225570	-0.871516	-0.163409	0.544698	1.960912
un_tam_cel	546.0	-8.743515e-17	1.0	-0.706795	-0.706795	-0.706795	0.566836	2.158875
un_forma_cel	546.0	9.800870e-17	1.0	-0.746659	-0.746659	-0.746659	0.549607	2.169940
adesao_marginal	546.0	9.190857e-17	1.0	-0.658547	-0.658547	-0.658547	0.372715	2.435239
tam_cel_epit	546.0	1.352195e-16	1.0	-0.998697	-0.563163	-0.563163	0.307905	2.921108
nucleos_exp	546.0	-4.636096e-17	1.0	-0.695540	-0.695540	-0.695540	0.612569	1.782983
cromatina	546.0	-3.700743e-17	1.0	-0.982966	-0.583636	-0.184306	0.215024	2.611003
nucleos_normais	546.0	2.257047e-16	1.0	-0.620388	-0.620388	-0.620388	0.347417	2.283027
mitoses	546.0	-9.007853e-17	1.0	-0.352816	-0.352816	-0.352816	4.716589	

### ▼ Exercício #3: Converter DataFrame para tf.Tensor

Para os dados poderem ser usados por uma RNA em um loop de treinamento customizado eles devem estar no forma de tensores do TensorFlow, assim, vamos converter os dados para tf.Tensor.

```
# Converte entradas para tf.Tensor
# Inclua seu código aqui
X_train = tf.convert_to_tensor(X_train_norm, dtype=tf.float32)
X_test = tf.convert_to_tensor(X_test_norm, dtype=tf.float32)

# Convert saídas para tf.Tensor e ajusta dimensões
# Inclua seu código aqui
Y_train = tf.reshape(tf.convert_to_tensor(train_Y.values, dtype=tf.int32), (-1, 1))
Y_test = tf.reshape(tf.convert_to_tensor(test_Y.values, dtype=tf.int32), (-1, 1))

print('Dimensão dos dados de entrada de treinamento:', X_train.shape)
print('Dimensão dos dados de entrada de teste:', X_test.shape)
```

```
print('Dimensão dos dados de saída de treinamento:', Y_train.shape)
print('Dimensão dos dados de saída de teste:', Y_test.shape)
```

→ Dimensão dos dados de entrada de treinamento: (546, 9)  
 Dimensão dos dados de entrada de teste: (137, 9)  
 Dimensão dos dados de saída de treinamento: (546, 1)  
 Dimensão dos dados de saída de teste: (137, 1)

**Saída esperada:**

```
Dimensão dos dados de entrada de treinamento: (546, 9)
Dimensão dos dados de entrada de teste: (137, 9)
Dimensão dos dados de saída de treinamento: (546, 1)
Dimensão dos dados de saída de teste: (137, 1)
```

```
print(X_train[:10])
print(Y_train[:10])

→ tf.Tensor(
[[ 0.2118086 -0.6997106 -0.74102694 -0.62850344 -0.5386761 -0.6975436
-0.15889214 -0.60249645 -0.34788916]
[-0.14512813 -0.6997106 -0.74102694  0.07614313 -0.9946717 -0.6975436
-0.5739884 -0.60249645 -0.34788916]
[-0.14512813  0.2951904  0.2764099  0.4284664  1.2853062  0.398453
 1.5014927  0.07160819 -0.34788916]
[-0.8590016 -0.6997106 -0.74102694 -0.62850344 -0.5386761 -0.6975436
-0.5739884 -0.26544413 -0.34788916]
[-1.2159383 -0.6997106 -0.74102694 -0.62850344 -0.9946717 -0.6975436
-0.5739884 -0.60249645 -0.34788916]
[-0.5020649 -0.03644326  0.9547011  0.4284664  0.82931066  1.2204504
 0.25620407  0.4086605 -0.34788916]
[ 0.5687454  0.6268241  0.2764099  0.4284664 -0.08268052  1.4944496
 1.5014927  1.7568698  0.8282562 ]
[-0.14512813 -0.6997106 -0.74102694 -0.62850344 -0.5386761 -0.6975436
-0.15889214 -0.60249645 -0.34788916]
[ 1.9964923  2.2849925  2.3112836  1.8377596  1.2853062 -0.6975436
 1.916589  2.0939221 -0.34788916]
[-1.2159383 -0.36807692 -0.74102694  0.07614313 -0.5386761 -0.6975436
-0.5739884 -0.60249645 -0.34788916]], shape=(10, 9), dtype=float32)
tf.Tensor(
[[0]
[0]
[0]
[0]
[0]
[1]
[1]
[0]
[1]
[0]], shape=(10, 1), dtype=int32)
```

**Saída esperada:**

```
tf.Tensor(
[[ 1.9609115  2.158875   1.8458735   0.02896096  1.6145062   0.4060258
-0.18430609  0.67001873 -0.35281572]
[-0.51746273 -0.70679533 -0.7466588 -0.658547  -0.56316274 -0.69553983
-0.98296577 -0.62038773 -0.35281572]
[-1.2255697 -0.70679533 -0.09852573 -0.658547  -0.56316274 -0.69553983
-0.98296577 -0.62038773 -0.35281572]
[-1.2255697 -0.70679533 -0.42259225 -0.658547  -0.56316274 -0.42014843
 0.21502376 -0.29778612 -0.35281572]
[-0.8715162  0.5668359   1.1977404   1.0602229  0.30790484  1.7829828
 1.4130133  0.99262035 -0.35281572]
[ 1.9609115  2.158875   1.1977404   1.7477309   1.6145062  -0.69553983
 2.611003  2.2830267  0.77371866]
[-1.2255697 -0.70679533 -0.7466588 -0.658547  -0.9986965 -0.69553983
-0.98296577  0.02481551 -0.35281572]
[ 0.19064417 -0.06997974 -0.42259225  1.7477309   0.74343866  1.7829828
 1.8123431  -0.62038773  0.21045148]
[ 1.9609115  0.24842808  0.54960734  0.37271494 -0.12762895  0.4060258
 1.4130133  0.02481551 -0.35281572]
[ 0.19064417 -0.70679533 -0.7466588 -0.658547  -0.56316274 -0.69553983
-0.18430609 -0.62038773 -0.35281572]], shape=(10, 9), dtype=float32)
tf.Tensor(
[[1]]
```

```
[0]
[0]
[0]
[1]
[1]
[0]
[1]
[1]
[0]], shape=(10, 1), dtype=int32)
```

## ▼ 4. Configuração e compilação da RNA

### Exercício #4: Configuração da RNA

Para realizar essa tarefa de classificação binária vamos utilizar uma RNA com 3 camadas tipo densa. Na célula abaixo configure a sua RNA usando os seguintes parâmetros:

- Primeira camada: 128 neurônios e função de ativação Relu;
- Segunda camada: 64 neurônios e função de ativação Relu;
- Camada de saída: 1 neurônio e função de ativação sigmoide.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

# Para você fazer: Configuração da RNA

# Inclua seu código aqui
rna = Sequential([
    Dense(128, activation='relu', input_shape=(X_train.shape[1],)),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Resumo da RNA
rna.summary()

→ /usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` arg
      super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"



| Layer (type)    | Output Shape | Param # |
|-----------------|--------------|---------|
| dense (Dense)   | (None, 128)  | 1,280   |
| dense_1 (Dense) | (None, 64)   | 8,256   |
| dense_2 (Dense) | (None, 1)    | 65      |



Total params: 9,601 (37.50 KB)
Trainable params: 9,601 (37.50 KB)
Non-trainable params: 0 (0.00 B)
```

### Saída esperada:

```
Model: "sequential"



| Layer (type)    | Output Shape | Param # |
|-----------------|--------------|---------|
| dense (Dense)   | (None, 128)  | 1280    |
| dense_1 (Dense) | (None, 64)   | 8256    |
| dense_2 (Dense) | (None, 1)    | 65      |



Total params: 9,601
Trainable params: 9,601
Non-trainable params: 0
```

## ▼ Exercício #5: Definição do otimizador da RNA, função de custo e métrica

Na célula abaixo defina o otimizador, a função de custo e a métrica que serão usados no treinamento da RNA.

- Otimizador: Adam com taxa de aprendizado de 0.001;
- Função de custo: BinaryCrossentropy
- Métrica: Accuracy

Observa-se que deve-se usar as versões na forma de classes de todas essas funções.

```
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import BinaryCrossentropy
from tensorflow.keras.metrics import BinaryAccuracy
# Para você fazer: definir otimizador, função de custo e métrica

# Define objeto otimizador usando tf.keras.optimizer.Adam
# Inclua seu código aqui
otimizador = Adam(learning_rate=0.001)

# Define objeto função de custo usando tf.keras.losses.BinaryCrossentropy
# Inclua seu código aqui
custo = BinaryCrossentropy()

# Define objeto métrica usando tf.keras.metrics.BinaryAccuracy
# Inclua seu código aqui
metrica = BinaryAccuracy()
```

Vamos avaliar a RNA não treinada para termos uma base do resultado esperado do treinamento. Execute as células a seguir para realizar essa avaliação.

```
loss_object = custo
metric_object = metrica

# Calcula previsões da RNA não treinada
outputs = rna.predict(X_test)

# Calcula função de custo
loss_value = loss_object(y_true=Y_test, y_pred=outputs)
print("Custo antes do treinamento =", loss_value.numpy())

# Calcula métrica
accuracy = metric_object(y_true=Y_test, y_pred=outputs)
print("Métrica antes do treinamento =", accuracy.numpy())
```

→ 5/5 0s 30ms/step  
Custo antes do treinamento = 0.6216349  
Métrica antes do treinamento = 0.79562044

**Saída esperada:**

```
Custo antes do treinamento = 0.766273
Métrica antes do treinamento = 0.16788322
```

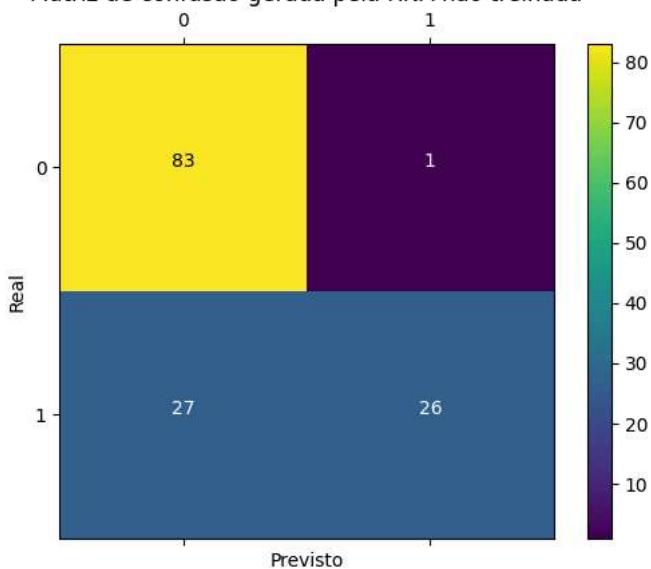
```
# Importa funções para calcular matriz e confusão
from sklearn.metrics import confusion_matrix
import itertools

# Define função para construir matriz de confusão
def plot_confusion_matrix(y_true, y_pred, title='', labels=[0,1]):
    cm = confusion_matrix(y_true, y_pred)
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cax = ax.matshow(cm)
    plt.title(title)
    fig.colorbar(cax)
    ax.set_xticklabels([''] + labels)
    ax.set_yticklabels([''] + labels)
    plt.xlabel('Previsto')
    plt.ylabel('Real')
    fmt = 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="black" if cm[i, j] > thresh else "white")
    plt.show()

plot_confusion_matrix(Y_test, tf.round(outputs), title='Matriz de confusão gerada pela RNA não treinada')
```

```
<ipython-input-22-bb5bb58a703c>:13: UserWarning: set_xticklabels() should only be used with a fixed number of ticks, i.e. after set_1
  ax.set_xticklabels([''] + labels)
<ipython-input-22-bb5bb58a703c>:14: UserWarning: set_yticklabels() should only be used with a fixed number of ticks, i.e. after set_1
  ax.set_yticklabels([''] + labels)
```

Matriz de confusão gerada pela RNA não treinada



## 5. Treinamento da RNA

Para treinar essa RNA vamos criar um loop customizado usando a função `tf.GradientTape()`.

**Exercício #6:** Criar função para calcular gradientes e atualizar parâmetros

Na célula abaixo crie um função que calcula o gradiente da função de custo em relação aos parâmetros da RNA e depois atualiza esses parâmetros usando o otimizador configurado anteriormente.

Para acessar os parâmetros de um modelo do TensorFlow basta usar `model.trainable_weights`.

```
# Para você fazer: criar função que calcula gradientes e atualiza parâmetros da RNA
def apply_gradient(optimizer, loss_object, model, x, y):
    ...
    Função para calcular o gradinet e atualizar os parâmetros da RNA

    Argumentos:
        optimizer: otimizador configurado para atualizar os parâmetros
        loss_object: função de custo configurada anteriormente
        model: RNA que está sendo treinada
        x: tensor com os dados de entrada de treinamento
        y: saídas desejadas dos dados de treinamento

    Retorna:
        logits = saídas previstas pela RNA
        loss_value = valor da função de custo
    ...

# Inclua seu código aqui
with tf.GradientTape() as tape:
    logits = model(x, training=True)
    loss_value = loss_object(y, logits)

gradients = tape.gradient(loss_value, model.trainable_weights)
optimizer.apply_gradients(zip(gradients, model.trainable_weights))

return logits, loss_value
```

Execute a célula abaixo para testar a sua função `apply_gradient()`.

```
otimizador = Adam(learning_rate=0.001)

# Cria nova RNA igual à configurada anteriormente
test_model = tf.keras.models.clone_model(rna)
test_model.set_weights(rna.get_weights())

# Calcula saída prevista e função de custo
test_logits, test_loss = apply_gradient(otimizador, loss_object, test_model, X_test, Y_test)
```

```
print('Primeiras 5 saídas:', test_logits.numpy()[:5])
print('\nFunção de custo =', test_loss.numpy())
```

```
del test_model
del test_logits
del test_loss
```

```
→ Primeiras 5 saídas: [[0.48142534]
 [0.4073954 ]
 [0.42870045]
 [0.453999 ]
 [0.49283117]]
```

Função de custo = 0.62163496

#### Saída esperada:

Primeiras 5 saídas: [[0.57383853]

```
[0.53157353]
[0.43801865]
[0.47596937]
[0.4938018 ]]
```

Função de custo = 0.6647787

### Exercício #7: Cálculo da função de custo e métrica para os dados de validação

No final de cada época de treinamento, temos que validar a RNA no conjunto de dados de teste. Crie uma função que calcula a função de custo e a métrica para os dados validação.

```
# Para você fazer: função para calcular custo e métrica dos dados de validação

# Função para calcular custo e métrica dos dados de validação
@tf.function()
def perform_validation(model, x_val, y_val):
    # Calcula custo dos dados de validação
    # Inclua seu código aqui
    val_logits = model(x_val, training=False)
    val_loss = loss_object(y_val, val_logits)

    # Calcula classes arredondando as saídas previstas (valores iguais a 0 ou 1)
    # Inclua seu código aqui
    val_predictions = tf.round(val_logits)

    # Calcula métrica para dados de validação
    # Inclua seu código aqui
    val_accuracy = metric_object(y_val, val_predictions)

    return val_loss, val_accuracy
```

Execute a célula abaixo para testar a sua função `perform_validation()`.

```
val_loss, val_accuracy = perform_validation(rna, X_test, Y_test)

print('Função de custo para os dados de teste =', val_loss.numpy())
print('Exatidão para os dados de teste =', val_accuracy.numpy())

→ Função de custo para os dados de teste = 0.62163496
Exatidão para os dados de teste = 0.79562044
```

#### Saída esperada:

```
Função de custo para os dados de teste = 0.7662731
Exatidão para os dados de teste = 0.16788322
```

### Exercício #8: Loop e treinamento customizado

Usando a função `apply_gradient()` vamos criar um loop de treinamento customizado. Utilize 1000 épocas de treinamento.

```
# Para você fazer: loop de treinamento customizado
otimizador = Adam(learning_rate=0.001)

# Define número de épocas
num_epocas = 1000

# Loop de treinamento
for i in range(num_epocas):
    # Calcula gradientes e atualiza parâmetros da RNA
    # Inclua seu código aqui
    logits, loss_value = apply_gradient(otimizador, loss_object, rna, X_train, Y_train)

    # Calcula métrica para dados de treinamento
    # Inclua seu código aqui
    accuracy = metric_object(y_true=Y_train, y_pred=tf.round(logits))

    # Calcula função de custo e métrica para dados de validação
    # Inclua seu código aqui
    val_loss, val_accuracy = perform_validation(rna, X_test, Y_test)

    # Imprime resultado da função de custo e métrica da época
    if i % 100 == 0:
        print('Época:', i, '-', 'custo =', loss_value.numpy(), '-', 'exatidão =', accuracy.numpy(), '-', 'custo_val =', val_loss.numpy(), '-')

    # Imprime resultado final
print('\nCusto final =', loss_value.numpy())
print('Exatidão final =', accuracy.numpy())
print('\nCusto final de validação =', val_loss.numpy())
print('Exatidão final de validação =', val_accuracy.numpy())

→ Época: 0 - custo = 0.6128442 - exatidão = 0.8219512 - custo_val = 0.5631743 - val_exatidão = 0.8328109
Época: 100 - custo = 0.062367514 - exatidão = 0.97008103 - custo_val = 0.050274473 - val_exatidão = 0.9701113
Época: 200 - custo = 0.03834365 - exatidão = 0.97670645 - custo_val = 0.059791062 - val_exatidão = 0.9767078
Época: 300 - custo = 0.016926488 - exatidão = 0.9806825 - custo_val = 0.08184999 - val_exatidão = 0.98068076
Época: 400 - custo = 0.006038599 - exatidão = 0.98424566 - custo_val = 0.1158039 - val_exatidão = 0.9842426
Época: 500 - custo = 0.0024991198 - exatidão = 0.9865126 - custo_val = 0.14569397 - val_exatidão = 0.98650926
Época: 600 - custo = 0.0013026602 - exatidão = 0.9880254 - custo_val = 0.16750294 - val_exatidão = 0.9880221
Época: 700 - custo = 0.0007850493 - exatidão = 0.9891067 - custo_val = 0.18437569 - val_exatidão = 0.9891036
Época: 800 - custo = 0.00051801355 - exatidão = 0.9899181 - custo_val = 0.19896407 - val_exatidão = 0.9899152
Época: 900 - custo = 0.00036291758 - exatidão = 0.9905462 - custo_val = 0.21180357 - val_exatidão = 0.9905418

Custo final = 0.00026898852
Exatidão final= 0.99090225

Custo final de validação = 0.22259179
Exatidão final de validação = 0.99089825
```

### Saída esperada:

```
Época: 0 - custo = 0.68436486 - exatidão = 0.34634146 - custo_val = 0.6501663 - val_exatidão = 0.3448276
Época: 100 - custo = 0.058527242 - exatidão = 0.9515191 - custo_val = 0.080492094 - val_exatidão = 0.9515428
Época: 200 - custo = 0.03760329 - exatidão = 0.96556544 - custo_val = 0.07935773 - val_exatidão = 0.9655706
Época: 300 - custo = 0.018621787 - exatidão = 0.9716702 - custo_val = 0.09142268 - val_exatidão = 0.97165996
Época: 400 - custo = 0.008029577 - exatidão = 0.9759616 - custo_val = 0.11785624 - val_exatidão = 0.97595173
Época: 500 - custo = 0.0036735435 - exatidão = 0.9789992 - custo_val = 0.14663258 - val_exatidão = 0.9789901
Época: 600 - custo = 0.0019131048 - exatidão = 0.9810311 - custo_val = 0.17128268 - val_exatidão = 0.98102283
Época: 700 - custo = 0.0011265007 - exatidão = 0.9824835 - custo_val = 0.19244863 - val_exatidão = 0.982476
Época: 800 - custo = 0.00072901906 - exatidão = 0.983407 - custo_val = 0.21075746 - val_exatidão = 0.9833984
Época: 900 - custo = 0.0005036661 - exatidão = 0.984111 - custo_val = 0.2269108 - val_exatidão = 0.98410314

Custo final = 0.0003456268
Exatidão final= 0.98466927

Custo final de validação = 0.24358612
Exatidão final de validação = 0.98466206
```

## ▼ 7. Avaliação e teste da RNA

### Exercício #9: Cálculo da função de custo e métrica para os dados de teste

Na célula abaixo calcule a função de custo e a métrica para os dados de teste.

```
#Para você fazer: cálculo da função de custo e métrica para os dados de test
```

```
# Calcula saída prevista para os dados de teste
# Inclua seu código aqui
```

```

outputs_test = rna(X_test, training=False)

# Calcula função de custo para os dados de teste
# Inclua seu código aqui
loss_value = loss_object(y_true=Y_test, y_pred=outputs_test)

# Calcula métrica para os dados de teste
# Inclua seu código aqui
metric_value = metric_object(y_true=Y_test, y_pred=tf.round(outputs_test))

print("Custo =", loss_value.numpy())
print("Exatidão -" metric_value.numpy())

```

Saída esperada:

```

Custo = 0.24358612
Exatidão = 0.9846549

```

Execute a célula abaixo para calcular a matriz de confusão para a RNA treinada.

```
plot_confusion_matrix(Y_test, tf.round(outputs), title='Matriz de confusão da RNA treinada')
```

```

→ <ipython-input-22-bb5bb58a703c>:13: UserWarning: set_ticklabels() should only be used with a fixed number of ticks, i.e. after set_1
  ax.set_xticklabels([''] + labels)
<ipython-input-22-bb5bb58a703c>:14: UserWarning: set_ticklabels() should only be used with a fixed number of ticks, i.e. after set_1
  ax.set_yticklabels([''] + labels)

```

