Aula 9

"Data Pipelines" - Parte 2

Etapa de treinamento

Eduardo Lobo Lustosa Cabral

1. Objetivos

Apresentar ferramentas do TensorFlow para criar "data pipelines" eficientes.

Apresentar formas de treinar RNAs usando ferramentas de "data pipelines" do TensorFlow.

Exemplos de treinamento de RNAs usando "data pipelines".

Importa principais bibliotecas

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

np.set_printoptions(precision=4)
```

2. Data Pipelines

Com já vimos, existem muitos tipos de dados e problemas. Como por exemplo:

- Imagens → classificação, detecção e localização de objetos, segmentação, geração de novas imagens.
- Texto → classificação, análise de sentimento, geração de novos textos, tradução de texto, chatbot.
- Áudio → reconhecimento de voz, música, geração de áudio (música e voz).
- Vídeo → classificação, reconhecimento de ação, rastreamento de objetos, entendimento de vídeo.
- Séries temporais → previsão, regressão (ajuste de função).
- Dados estruturados → regressão, sistemas de recomendação, classificação.

Para processar e carregar dados para treinar uma RNA de forma eficiente deve-se criar um "data pipeline".

Data pipelines funcionam no princípio ETC (Extrair, Transformar e Carregar) que em inglês é ETL (Extraction, Tranformation and Load).

- **Extrair** → carrega dados originais do local onde se encontram e traz para o nosso ambiente de computação;
- **Transformar** → processa os dados para serem colocados em formatos adequados que possam ser usados por uma RNA;
- **Carregar** → alimenta a RNA com dados durante o seu treinamento ou para realizar previsões quando colocada em operação.

Cada tipo de dado exige um "data pipeline" diferente. Por exemplo:

- Imagens → ler arquivos, aplicar transformações em cada imagem e juntar aletoriamente em lotes para treinamento.
- Texto → ler arquivos, pode envolver extrair palavras ou letras do texto, converter em vetores “one-hote” ou “embeding” e criar lotes de sequências que podem ter comprimentos diferentes.
- Vídeo → ler arquivos, separar imagens dos vídeos, aplicar transformações em cada imagem e juntar em lotes de treinamento que podem ter comprimentos diferentes.
- Áudio e séries temporais → ler arquivos, criar janelas com dados temporais, aplicar transformações nos dados, juntar em lotes de treinamento.
- Dados estruturados → ler arquivos, transformar dados e juntar em lotes para treinamento.

"Data pipelines" com TensorFlow

O TensorFlow fornece ferramentas para realizar as três etapas de um "data pipeline" de forma eficiente para qualquer tipo de dado e problema.

A grande vantagem de usar as ferramentas do TensorFlow é que elas são otimizadas para funcionar com os métodos de treinamento do Keras e, assim, o processo de treinamento é mais rápido.

O módulo **`tf.data`** do TensorFlow disponibiliza ferramentas para criar "data pipelines" complexos de forma simples (https://www.tensorflow.org/api_docs/python/tf/data).

Na Aula 4 vimos as etapas de "extração" e "transformação" dos dados → **nessa aula veremos a etapa de carregar os dados no treinamento de uma RNA.**

3. Treinamento

A última etapa de um "data pipeline" de dados para desenvolver uma RNA é carregar os dados durante o processo de treinamento.

Com os dados em um objeto Dataset, eles são fornecidos para o treinamento da RNA de forma otimizada utilizando da melhor forma possível todos os recursos de CPU/GPU e memória disponíveis.

Com um Dataset é possível utilizar os métodos `fit()`, `evaluate()` e `predict()` do Keras.

Como exemplos de usar um conjunto de dados no formato Dataset no treinamento de uma RNA, vamos mostrar dois casos:

1. Dados na forma de imagens em tensores para classificação multiclasse;
2. Dados estruturados em arquivo CSV para classificaçao binária com classes desbalanceadas.

4. Classificação multiclasse com dados em imagens

Nesse exemplos vamos utilizar os dados do conjunto Fashion-MNIST.

Como já vimos o conjunto de dados Fashion-MNIST consiste de imagens de artigos de vesturário divididas em 10 classes.

4.1 Carregar dados

O código abaixo carrega os dados da Fashion-MNIST da coleção do Keras e separa as imagens de entrada e as classes.

```
# Carrega conjunto de dados do keras
train, test = tf.keras.datasets.fashion_mnist.load_data()

# Separa imagens e classes
images_train, labels_train = train
images_test, labels_test = test

# Mostra dimensões dos dados
print('Dimensão do tensor de imagens de treinamento:',
images_train.shape)
print('Dimensão do tensor de imagens de teste:', images_test.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 ━━━━━━━━━━━━━━━━━━━━ 0s 1us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
Dimensão do tensor de imagens de treinamento: (60000, 28, 28)
Dimensão do tensor de imagens de teste: (10000, 28, 28)
```

Pode-se ver que esse conjunto de dados possui 60.000 exemplos de treinamento e 10.000 exemplos de teste. Além disso, as imagens tem 28x28 pixels e são em tons de cinza.

4.2 Criar Datasets de treinamento e teste

Na célula abaixo são criados os Datasets de treinamento e teste.

```
# Cria datasets de treinameto e teste
fmnist_train_ds = tf.data.Dataset.from_tensor_slices((images_train,
labels_train))
fmnist_test_ds = tf.data.Dataset.from_tensor_slices((images_test,
labels_test))
```

Para transformar os dados de forma a que possam ser usados por uma RNA, vamos definir a função `transform()`, que realiza as seguintes operações ao carregar os lotes de dados:

1. Redefine o tipo dos dados das imagens para serem reais;
2. Normaliza as imagens para os pixels terem valores entre 0 e 1;
3. Codifica as classes das imagens para vetores one-hot.

```
## Define função para normalizar as imagens codificar saída
def transform(x, y):
    x_norm = tf.cast(x, dtype=tf.float32)/255.
    y_int = tf.cast(y, dtype=tf.int32)
    y_hot = tf.one_hot(y_int, 10)
    return x_norm, y_hot
```

- Observe que essa função recebe tanto as entradas como as saídas dos exemplos de treinamento em razão dos Datasets retornarem pares imagem-classe.

Agora vamos introduzir a transformação dos dados nos Datasets de treinamento e teste criados anteriormente. Para isso usamos o método `map` e a função `transform()`.

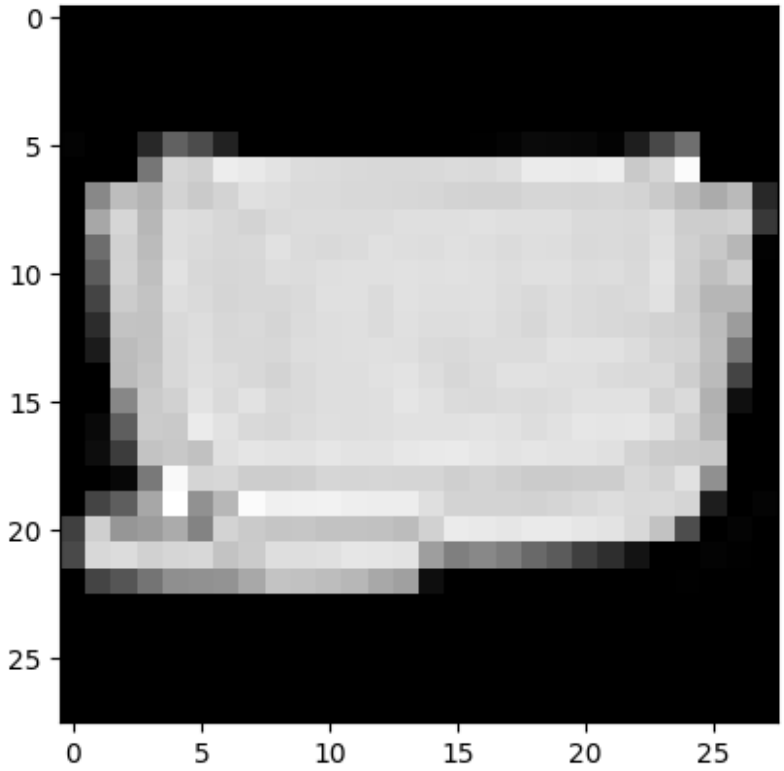
```
# Define tamanho do lote
batch_size = 32

# Cria Dataset com a transformação que normaliza as imagens e codifica
saída
fmnist_train_ds = fmnist_train_ds.map(transform,
num_parallel_calls=tf.data.AUTOTUNE)
fmnist_test_ds = fmnist_test_ds.map(transform,
num_parallel_calls=tf.data.AUTOTUNE)

# Define tamanho de lotes e embaralha dados
fmnist_train_ds = fmnist_train_ds.shuffle(1000).batch(batch_size)
fmnist_test_ds = fmnist_test_ds.shuffle(1000).batch(batch_size)
```

- Observe que esses Datasets retornam um lote de imagens normalizadas e saídas em vetores one-hot. Para realizar essas transfromações é usado o método `map` que chama a função `transform()`.

Para verificar se os Datasets estão funcionando corretamente, vamos criar um lote e apresentar alguns exemplos

```
# Gera um lote de treinamento
img, y = next(iter(fmnist_train_ds))

# Mostra primeiro exemplo do lote
print('Classe:', y[0])
plt.imshow(img[0], cmap='gray')
plt.show()

# Apresenta dimensão do lote de imagens e alguns pixels da primeira
imagem
print('Dimensão do lote:', img.shape)
print('Valores de alguns pixels:', img[0,14,10:15].numpy())
```

```
Classe: tf.Tensor([0. 0. 0. 0. 0. 0. 0. 0. 1. 0.], shape=(10,),
dtype=float32)
```




```
Dimensão do lote: (32, 28, 28)
Valores de alguns pixels: [0.8706 0.8706 0.8824 0.8941 0.8667]
```

4.3 Configuração e compilação da RNA

Como as imagens tem dimensão pequena (28x28) e são em tons de cinza, vamos usar uma RNA simples com duas camadas densas para resolver esse problema.

A RNA é compilada com os seguintes parâmetros:

- Método de otimização: Adam com a sua taxa de aprendizado padrão (`lr=0.001`);
- Função de custo: `CategoricalCrossentropy`;
- Métrica: `accuracy`.

```
# Define dimensão das imagens
img_size = (28, 28)

# Cria RNA
rna = tf.keras.Sequential([
      tf.keras.layers.Flatten(input_shape=img_size),
      tf.keras.layers.Dense(32, activation='relu'),
      tf.keras.layers.Dense(10, activation='softmax')])

# Apresenta sumário da RNA
rna.summary()

# Compila RNA
rna.compile(optimizer='adam',
            loss= tf.keras.losses.CategoricalCrossentropy(),
            metrics=['accuracy'])
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/reshaping/
flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_dim`
argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

```
Model: "sequential_1"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━
┓
┃ Layer (type)                         ┃ Output Shape                ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━
┩
│ flatten_1 (Flatten)                  │ (None, 784)                 │
0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────
┤
│ dense_2 (Dense)                      │ (None, 32)                  │
25,120 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────
┤
│ dense_3 (Dense)                      │ (None, 10)                  │
330 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────
┘

 Total params: 25,450 (99.41 KB)

 Trainable params: 25,450 (99.41 KB)
```

```
 Non-trainable params: 0 (0.00 B)
```

- A primeira camada da RNA é uma camada tipo `Flatten` para redimensionar as imagens e transformá-las em um vetor para poderem ser processadas por uma camada densa. Observe que o redimensionamento das imagens foi incluído dentro da RNA.

4.4 Treinamento da RNA

Para treinar a RNA basta passar os Datasets de treinamento e teste da mesma forma como se estivessem em tensores.

Somente para exemplificar, vamos usar poucas épocas de treinamento.

```
rna.fit(fmnist_train_ds, epochs=10, validation_data=fmnist_test_ds)
```

```
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.7551 - loss:
0.7304 - val_accuracy: 0.8395 - val_loss: 0.4654
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8521 - loss:
0.4234 - val_accuracy: 0.8464 - val_loss: 0.4335
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 3ms/step - accuracy: 0.8617 - loss:
0.3868 - val_accuracy: 0.8521 - val_loss: 0.4189
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8682 - loss:
0.3688 - val_accuracy: 0.8502 - val_loss: 0.4166
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8733 - loss:
0.3524 - val_accuracy: 0.8371 - val_loss: 0.4624
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8777 - loss:
0.3402 - val_accuracy: 0.8599 - val_loss: 0.3897
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8812 - loss:
0.3279 - val_accuracy: 0.8612 - val_loss: 0.3895
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.8838 - loss:
0.3195 - val_accuracy: 0.8670 - val_loss: 0.3841
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - accuracy: 0.8869 - loss:
0.3094 - val_accuracy: 0.8616 - val_loss: 0.3886
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.8896 - loss:
0.3015 - val_accuracy: 0.8643 - val_loss: 0.3915

<keras.src.callbacks.history.History at 0x78135e4efe50>
```

Importante:

Como visto, um Dataset criado usando o método `repeat` gera infinitos exemplos de treinamento. Nesses casos, no treinamento deve-se usar o argumento `steps_per_epoch` para definir quantos lotes por época são desejados.

4.5 Avaliação da RNA

O método `evaluate` pode ser usado com os dados em um Dataset.

Nesse caso pode-se se quiser, ou se for necessário, definir o número de lotes usados no treinamento com o argumento `steps`.

```
# Calcula função de custo e exatidão para todos os dados de teste
loss, accuracy = rna.evaluate(fmnist_test_ds)

print("Função de custo:", loss)
print("Exatidão:", accuracy)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8669 - loss: 0.3870
Função de custo: 0.39153170585632324
Exatidão: 0.864300012588501
```

```
# Calcula função de custo e exatidão para 10 lotes dos dados de teste
loss, accuracy = rna.evaluate(fmnist_train_ds.repeat(), steps=10)

print("Função de custo:", loss)
print("Exatidão:", accuracy)
```

```
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8811 - loss: 0.3311
Função de custo: 0.29792526364326477
Exatidão: 0.8843749761581421
```

4.6 Realização de previsões

Para usar o método `predict` para fazer previsões com a RNA as saídas não são necessárias. Assim, pode-se fazer o seguinte:

1. Criar um novo Dataset com somente as imagens de teste;
2. Usar o Dataset criado anteriormente como os dados de teste, que gera tanto as imagens como as classes, nesse caso as sáidas são ignoradas pelo método `predict`.

```
# Define função para somente normalizar imagens
def img_norm(x):
    x = tf.cast(x, dtype=tf.float32)/255.
    return x

# Cria novo Dataset somente com as imagens de teste sem as classes
predict_ds = tf.data.Dataset.from_tensor_slices(images_test)
predict_ds = predict_ds.map(img_norm).batch(batch_size)
```

```
# Realiza previsões
result = rna.predict(predict_ds, steps=10)

# Mostra dimensão dos resultados
print('Dimensão do tensor de previsões:', result.shape)
print('Probabilidades das classes do primeiro exemplo:', result[0])
```

```
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
Dimensão do tensor de previsões: (320, 10)
Probabilidades das classes do primeiro exemplo: [9.6295e-07 1.2638e-10
1.5974e-06 5.0832e-06 5.5423e-07 1.7829e-01
 8.0083e-05 8.4911e-02 2.2027e-03 7.3450e-01]
```

- Observe que as saídas da RNA são vetores, cujos elementos representam as probabilidades da imagem ser uma das classes.

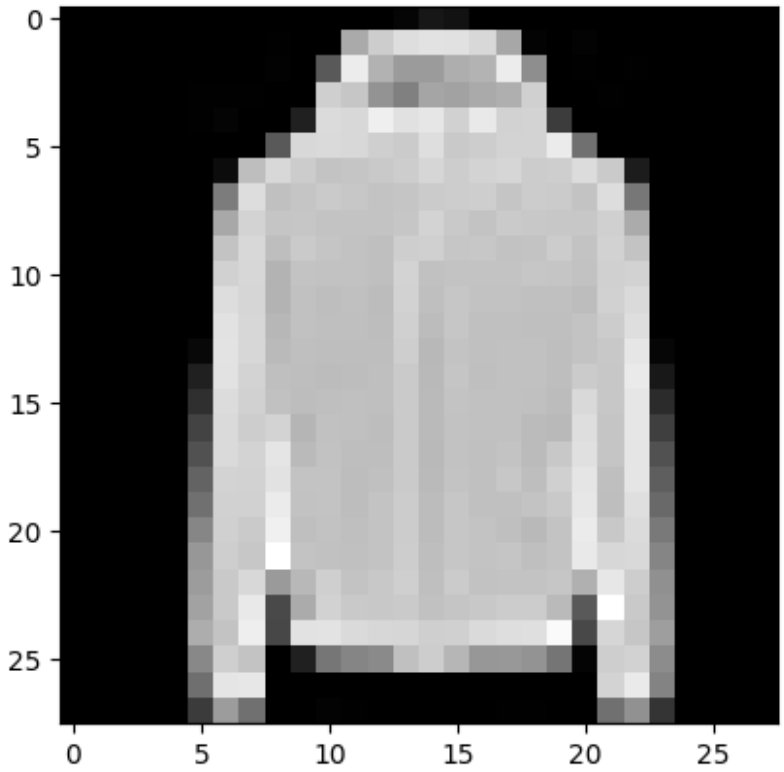
Se quisermos verificar os resultados comparando com as classes reais, é mais fácil criar um lote de imagens e classes e depois usar a RNA com o método `predict` passando somente as imagens do lote.

```
# Gera lote de imagens e saídas reais
img, y_real = next(iter(fmnist_test_ds))

# Determina categoria da classe real
classe_real = np.argmax(y_real, axis=1)

# Calcula previsão da RNA para um lote de imagens
y_prev = rna.predict(img)
classe_prev = np.argmax(y_prev, axis=1)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step
```

```
# Seleciona exemplo para mostrar resultado
index = 31

# Mostra um exemplo do lote
print('Classe real:', classe_real[index])
print('Classe prevista:', classe_prev[index])
print('Probabilidades das classe:' , y_prev[index])
plt.imshow(img[index], cmap='gray')
plt.show()
```

```
Classe real: 4
Classe prevista: 4
Probabilidades das classe: [1.8467e-04 5.9024e-07 5.8754e-02 1.9999e-
04 9.3494e-01 7.1425e-11
 5.9123e-03 2.5078e-10 1.2947e-05 9.6232e-16]
```



5. Classificação binária com classes desbalanceadas - dados estruturados

Nesse exemplo vamos utilizar os dados do conjunto "The credit card fraud".

Como já vimos na aula passada, esse conjunto de dados consiste de um problema de classificação binária com dados desbalanceados, onde os dados sem fraude (classe = 0) representam 99,6% do total e os dados com fraude (classe=1) representam 0,4%.

5.1 Carregar os dados

Vamos carregar os dados para podermos visualizá-los diretamente do arquivo onde se encontram.

Nas céulas abaixo é definido o local onde se encontra os dados e depois os dados são carregados e descompactados.

```
df =
pd.read_csv('https://storage.googleapis.com/download.tensorflow.org/
data/creditcard.csv')
df
```

```
{"type":"dataframe","variable_name":"df"}
```

Características dos dados:

- Existem 284.807 exemplos de treinamento (cada linha é um exemplo);
- Cada exemplo é composto por um vetor de entrada com 30 características e uma saída;
- As 30 características são as primeiras 30 colunas dos dados;
- As saídas estão na última coluna, de nome "Class".

Vamos verificar os tipos de dados de cada coluna. Isso é importante porque se existir algum dado na forma de string ele tem que ser transformado para real.

```
df.dtypes
```

```
Time      float64
V1        float64
V2        float64
V3        float64
V4        float64
V5        float64
V6        float64
V7        float64
V8        float64
V9        float64
V10       float64
V11       float64
V12       float64
V13       float64
V14       float64
V15       float64
V16       float64
V17       float64
V18       float64
V19       float64
V20       float64
V21       float64
V22       float64
V23       float64
V24       float64
V25       float64
V26       float64
V27       float64
V28       float64
Amount    float64
Class       int64
dtype: object
```

Vamos calcular a estatísticas das colunas para verificar como devemos pré-processar cada elemento dos exemplos de treinamento.

```
df.describe().T
```

```
{"summary":"{\n  \"name\": \"df\",\n  \"rows\": 31,\n  \"fields\": [\n{\n      \"column\": \"count\",\n      \"properties\": {\n\"dtype\": \"number\",\n        \"std\": 0.0,\n        \"min\":\n284807.0,\n        \"max\": 284807.0,\n        \"num_unique_values\":\n1,\n        \"samples\": [\n          284807.0\n        ],\n\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"mean\",\n      \"properties\": {\n\"dtype\": \"number\",\n        \"std\": 17028.550324734668,\n\"min\": -2.4063305498905906e-15,\n        \"max\":\n94813.85957508067,\n        \"num_unique_values\": 31,\n\"samples\": [\n          -3.6600908126037946e-16\n        ],\n\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"std\",\n      \"properties\": {\n\"dtype\": \"number\",\n        \"std\": 8527.578758378544,\n\"min\": 0.0415271896355952,\n        \"max\": 47488.14595456582,\n\"num_unique_values\": 31,\n        \"samples\": [\n0.4036324949650267\n        ],\n        \"semantic_type\": \"\",\n\"description\": \"\"\n      }\n    },\n    {\n      \"column\":\n\"min\",\n      \"properties\": {\n        \"dtype\": \"number\",\n\"std\": 26.795994690128563,\n        \"min\": -113.743306711146,\n\"max\": 0.0,\n        \"num_unique_values\": 29,\n\"samples\": [\n          -22.5656793207827\n        ],\n\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"25%\",\n      \"properties\": {\n\"dtype\": \"number\",\n        \"std\": 9734.925083048505,\n\"min\": -0.920373384390322,\n        \"max\": 54201.5,\n\"num_unique_values\": 31,\n        \"samples\": [\n          -\n0.07083952930446921\n        ],\n        \"semantic_type\": \"\",\n\"description\": \"\"\n      }\n    },\n    {\n      \"column\":\n\"50%\",\n      \"properties\": {\n        \"dtype\": \"number\",\n\"std\": 15211.001352467909,\n        \"min\": -0.274187076506651,\n\"max\": 84692.0,\n        \"num_unique_values\": 31,\n\"samples\": [\n          0.0013421459786502\n        ],\n\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"75%\",\n      \"properties\": {\n\"dtype\": \"number\",\n        \"std\": 25022.157936531003,\n\"min\": 0.0,\n        \"max\": 139320.5,\n\"num_unique_values\": 31,\n        \"samples\": [\n0.09104511968580689\n        ],\n        \"semantic_type\": \"\",\n\"description\": \"\"\n      }\n    },\n    {\n      \"column\":\n\"max\",\n      \"properties\": {\n        \"dtype\": \"number\",\n\"std\": 31218.887137498223,\n        \"min\": 1.0,\n        \"max\":\n172792.0,\n        \"num_unique_values\": 31,\n        \"samples\": [\n          31.6121981061363\n        ],\n        \"semantic_type\":\n\"\",\n        \"description\": \"\"\n      }\n    }\n  ]\n}","type":"dataframe"}
```

5.2 Pré-processamento dos dados

Para pré-processador os dados temos que fazer uma análise cuidadosa de cada característica (coluna) individualmente. Contudo, vamos realizar uma análise bastante simplificada, pois analisar dados não é o objetivo dessa aula.

Analisando cada uma das colunas de características temos que:

1. A primeira coluna, de nome "Time", não parece ter nenhuma informação relevante, assim, podemos tirá-la dos dados;

2. As colunas de nomes "V1" a "V28" são valores reais que variam de negativo a positivo e todas possuem média praticamente igual a 0 e desvio padrão igual a 1. Então vamos deixá-las como estão mesmo que alguns valores sejam da ordem de 10^1 .

3. A coluna "Amount" possui somente valores positivos, então podemos simplesmente dividi-la pelo seu valor máximo ou pelo seu desvio padrão.

Podemos realizar o pré-processamento dos dados previamente, ou podemos realizar esse processsamento ao obter os lotes de dados durante o treinamento → nesse exemplo vamos realizar o pré-processamento com o Pandas.

Embaralhamento aleatório dos dados

Para evitar qualquer tendência existente nos dados vamos embaralhá-los aleatóriamente.

```
df_shuffled=df.sample(frac=1)
df_shuffled
```

```
{"type":"dataframe","variable_name":"df_shuffled"}
```

Separar as saídas desejadas

Se vamos criar um Dataset a partir de um DataFrame Pandas devemos separar as entradas das saídas.

```
# Separa o vetor de saídas desejadas
y = df_shuffled.Class
y
```

```
177038    0
218734    0
144298    0
49840     0
154546    0
         ..
178674    0
246064    0
256292    0
```

```
90006     0
256447    0
Name: Class, Length: 284807, dtype: int64
```

Remover as colunas desnecessárias

Tendo o vetor de saídas desejadas, agora devemos remover as colunas "Time", que não tem informação nenhuma, e a coluna das saídas ("Class").

```
# remoção da coluna "Time" e da saída
df_shuffled.drop(['Time', 'Class'], axis=1, inplace=True)
df_shuffled
```

{"type":"dataframe","variable_name":"df_shuffled"}

Normalizar a coluna "Amount"

```
x = df_shuffled.copy()
x["Amount"] = x["Amount"]/x.Amount.max()

x.describe()
```

{"type":"dataframe"}

5.3 Criar objeto Dataset

Tendo os dados de entrada e de saída em um DataFrame Pandas, usamos o método `from_tensor_slices` para criar o Dataset.

Para poder usar os dados do Dataframe, devemos tirar os nomes das colunas e passar somente os valores numéricos dos dados. Isso é feito usando a propriedade `values`, da forma apresentada na célula a seguir.

```
# Cria Dataset
creditcard_ds = tf.data.Dataset.from_tensor_slices((x.values,
y.values))
```

- Observe que os valores numéricos do Dataframe são passados para o Dataset usando `x.values` e `y.values`.

Vamos gerar um elemento do Dataset para verificar o resultado.

```
for element in creditcard_ds.take(1):
    print(format(element))
```

```
(<tf.Tensor: shape=(29,), dtype=float64, numpy=
array([-6.2693e-01,  7.3248e-01, -3.3474e-01, -1.4769e+00,
1.5165e+00,
       -5.7300e-01,  9.0096e-01,  8.7592e-02,  4.8023e-01, -
2.3752e+00,
```

```
        5.5056e-02, -3.4257e-01, -1.3860e+00, -2.0551e+00, -
1.0411e+00,
        5.1915e-01,  9.8445e-01,  1.8356e+00,  2.1451e-01, -8.5495e-
02,
       -5.2248e-02, -7.7354e-02, -7.3005e-01, -1.2992e+00,
1.4469e+00,
       -4.5120e-01, -4.4104e-02, -4.3292e-02,  5.8386e-04])>,
<tf.Tensor: shape=(), dtype=int64, numpy=0>)
```

Divisão dos dados nos conjuntos de treinamento e teste

Vamos dividir os dados nos conjuntos de treinamento e teste, de forma a ter 80% dos dados no conjunto de treinamento e 20% no conjunto de teste.

```
# Define tamanho do lote
batch_size = 4096

# Define função que seleciona exemplos de teste
def is_test(x, y):
    return x % 5 == 0

# Define função que seleciiona exemplos de treinamento
def is_train(x, y):
    return not is_test(x, y)

# Define função que elimina índice usado para selecionar exemplos
incluidos com o método enumerate
recover = lambda x,y: y

# Cria Dataset de teste
test_ds = creditcard_ds.enumerate().filter(is_test).map(recover,
num_parallel_calls=tf.data.AUTOTUNE).batch(batch_size)

# Cria Dataset de treinamento
train_ds = creditcard_ds.enumerate().filter(is_train).map(recover,
num_parallel_calls=tf.data.AUTOTUNE).batch(batch_size)
```

Vamos gerar um exemplo de treinamento e um de teste para verificar se os Datasets estão corretos.

```
# Verifica Dataset de treinamento
for element in train_ds.take(1):
    print('Exemplo de treinamento:\m', format(element))

print(' ')

# Verifica Dataset de teste
for element in test_ds.take(1):
    print('Exemplo de teste:\n', format(element))
```

```
Exemplo de treinamento:\m (<tf.Tensor: shape=(4096, 29),
dtype=float64, numpy=
array([[ 1.9748e+00,  2.3764e-01, -1.5942e+00, ..., -1.1291e-02,
        -5.4208e-02,  1.4402e-03],
       [-1.4982e+00,  1.5623e+00, -1.5434e-02, ..., -4.9556e-01,
        -1.7440e-01,  6.9479e-04],
       [-6.7010e+00, -4.5265e+00, -5.3984e-01, ...,  1.1078e+00,
        -2.4468e+00,  4.7370e-04],
       ...,
       [ 9.4575e-01, -8.5752e-01,  8.6141e-01, ..., -4.6790e-02,
         1.3905e-02,  4.4046e-03],
       [-1.7907e+00,  1.9897e+00,  2.2296e-01, ...,  3.2777e-01,
         1.3427e-01,  3.4954e-04],
       [ 1.3490e+00, -4.0604e-01,  6.3085e-01, ...,  6.8708e-03,
         1.5644e-02,  4.4062e-04]])>, <tf.Tensor: shape=(4096,),
dtype=int64, numpy=array([0, 0, 0, ..., 0, 0, 0])>)

Exemplo de teste:
 (<tf.Tensor: shape=(4096, 29), dtype=float64, numpy=
array([[-6.2693e-01,  7.3248e-01, -3.3474e-01, ..., -4.4104e-02,
        -4.3292e-02,  5.8386e-04],
       [ 1.2895e+00, -8.4933e-01,  9.5938e-01, ...,  4.1626e-02,
         9.2966e-03,  4.2816e-04],
       [-1.1155e+00,  8.5202e-01, -1.5982e-01, ..., -7.8297e-02,
         7.7038e-03,  4.6670e-04],
       ...,
       [ 1.7887e+00, -9.0559e-01, -1.3119e+00, ...,  3.4903e-02,
        -2.4293e-02,  3.5308e-03],
       [-5.0567e-01,  7.8183e-01,  4.0118e-01, ..., -2.0708e-01,
        -1.8308e-01,  6.9674e-05],
       [-5.8282e-01, -1.0838e+00,  1.0900e+00, ...,  4.3642e-02,
         1.3089e-01,  3.5343e-03]])>, <tf.Tensor: shape=(4096,),
dtype=int64, numpy=array([0, 0, 0, ..., 0, 0, 0])>)
```

Criar Dataset que gera lotes com número de classes balanceadas

Para criar lotes com número de exemplos balanceados entre as duas classes vamos utilizar a abordagem de amostragem do Dataset. Assim, primeiramente vamos criar os dois Datasets, uma para cada classe, para depois criar o Dataset que gera lotes balanceados.

O ideal é fazer esse balanceamento para os conjuntos de treinamento e teste se formos usar os dados de teste para validação.

```
# Cria Datasets com os dados sem fraude (negativo, classe = 0)
negativos_ds_train = train_ds.unbatch().filter(lambda features, label:
label==0).repeat()
negativos_ds_test = test_ds.unbatch().filter(lambda features, label:
label==0).repeat()
```

```
# Cria Datasets com os dados com fraude (positivo, classe = 1)
positivos_ds_train = train_ds.unbatch().filter(lambda features, label:
label==1).repeat()
positivos_ds_test = test_ds.unbatch().filter(lambda features, label:
label==1).repeat()

# Cria Datasets que gera lotes de elementos com as classes balanceadas
balanced_ds_train = tf.data.experimental.sample_from_datasets(
                    [negativos_ds_train, positivos_ds_train], [0.5,
0.5]).batch(batch_size)

balanced_ds_test = tf.data.experimental.sample_from_datasets(
                    [negativos_ds_test, positivos_ds_test], [0.5,
0.5]).batch(batch_size)
```

```
WARNING:tensorflow:From <ipython-input-13-97a5b97d6a6a>:10:
sample_from_datasets_v2 (from
tensorflow.python.data.experimental.ops.interleave_ops) is deprecated
and will be removed in a future version.
Instructions for updating:
Use `tf.data.Dataset.sample_from_datasets(...)`.
```

Geração de um lote para verificação.

```
for features, labels in balanced_ds_train.take(1):
    print(features)
    print(labels)
```

```
tf.Tensor(
[[-2.3493e+00  1.5126e+00 -2.6475e+00 ... -7.3607e-01  7.3370e-01
   1.9073e-04]
 [ 1.2326e+00 -5.4893e-01  1.0879e+00 ...  8.0805e-02  3.5427e-02
   7.6252e-04]
 [-1.5192e+01  1.0433e+01 -1.9630e+01 ... -2.6348e+00 -4.6393e-01
   3.8924e-05]
 ...
 [-3.8912e+00  7.0989e+00 -1.1426e+01 ...  1.8815e+00  8.7526e-01
   3.8924e-05]
 [-7.0938e-01  1.1943e-01  2.0099e+00 ... -1.0448e-01 -1.3100e-01
   3.8924e-04]
 [ 1.6597e+00 -1.3131e+00 -1.1913e+00 ... -1.0266e-01 -2.6565e-02
   9.2736e-03]], shape=(4096, 29), dtype=float64)
tf.Tensor([1 1 1 ... 1 0 0], shape=(4096,), dtype=int64)
```

Vamos verificar o resultado dessas operações verificando o número de exemplos de cada classe de um lote.

```
# Gera 1 lote do Dataset balanced_ds
for features, labels in balanced_ds_train.take(2):
```

```
    n1 = tf.reduce_sum(labels)
    print('Número de exemplo com fraude (classe=1):', n1.numpy())
    print('Número de exemplo sem fraude (classe=0):', (batch_size - 
n1).numpy())
    print(' ')
```

```
Número de exemplo com fraude (classe=1): 2077
Número de exemplo sem fraude (classe=0): 2019

Número de exemplo com fraude (classe=1): 2045
Número de exemplo sem fraude (classe=0): 2051
```

- Pode-se ver que o número de exemplos de cada classe é quase o mesmo em todos os lotes gerados, girando em torno de 512, que é metade do tamanho do lote de 1024 exemplos.

5.4 Configuração e compilação da RNA

Para resolver esse problema vamos utilizar uma RNA com duas camadas densas, com a configuração a seguir.

```
# Importa classes
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Sequential

# Configura RNA
rna = Sequential()
rna.add(Dense(128, activation='relu', input_shape=(29,)))
rna.add(Dense(64, activation='relu'))
rna.add(Dense(1, activation='sigmoid'))

# Apresenta sumário 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`
argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)

Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape               ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
━━━━━━━━━━━━━━━━┩
```

```
| dense (Dense)                        | (None, 128)                 |
3,840 |
├──────────────────────────────────────┼─────────────────────────────┼
──────────────┤
| dense_1 (Dense)                      | (None, 64)                  |
8,256 |
├──────────────────────────────────────┼─────────────────────────────┼
──────────────┤
| dense_2 (Dense)                      | (None, 1)                   |
65 |
└──────────────────────────────────────┴─────────────────────────────┴
──────────────┘

 Total params: 12,161 (47.50 KB)

 Trainable params: 12,161 (47.50 KB)

 Non-trainable params: 0 (0.00 B)
```

Para compilar a RNA usaremos os seguintes parâmetros:

- Método de otimização: Adam
- Taxa de aprendizado: 0.001
- Métrica: exatidão

```
# Compila RNA
rna.compile(optimizer=tf.keras.optimizers.Adam(0.001),
            loss='binary_crossentropy',
            metrics=['accuracy'])
```

5.5 Treinamento da RNA

Para treinar a RNA basta passar os Datasets de treinamento e teste.

Somente para exemplificar, vamos usar poucas épocas de treinamento.

```
results = rna.fit(balanced_ds_train.repeat(),
                  steps_per_epoch=20,
                  epochs=10,
                  validation_data=balanced_ds_test.repeat(),
                  validation_steps=1,
                  verbose=1)

Epoch 1/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 1052s 53s/step - accuracy: 0.6628 - loss:
0.8265 - val_accuracy: 0.9380 - val_loss: 0.2215
Epoch 2/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 867s 46s/step - accuracy: 0.9318 - loss:
0.2140 - val_accuracy: 0.9490 - val_loss: 0.1446
Epoch 3/10
```

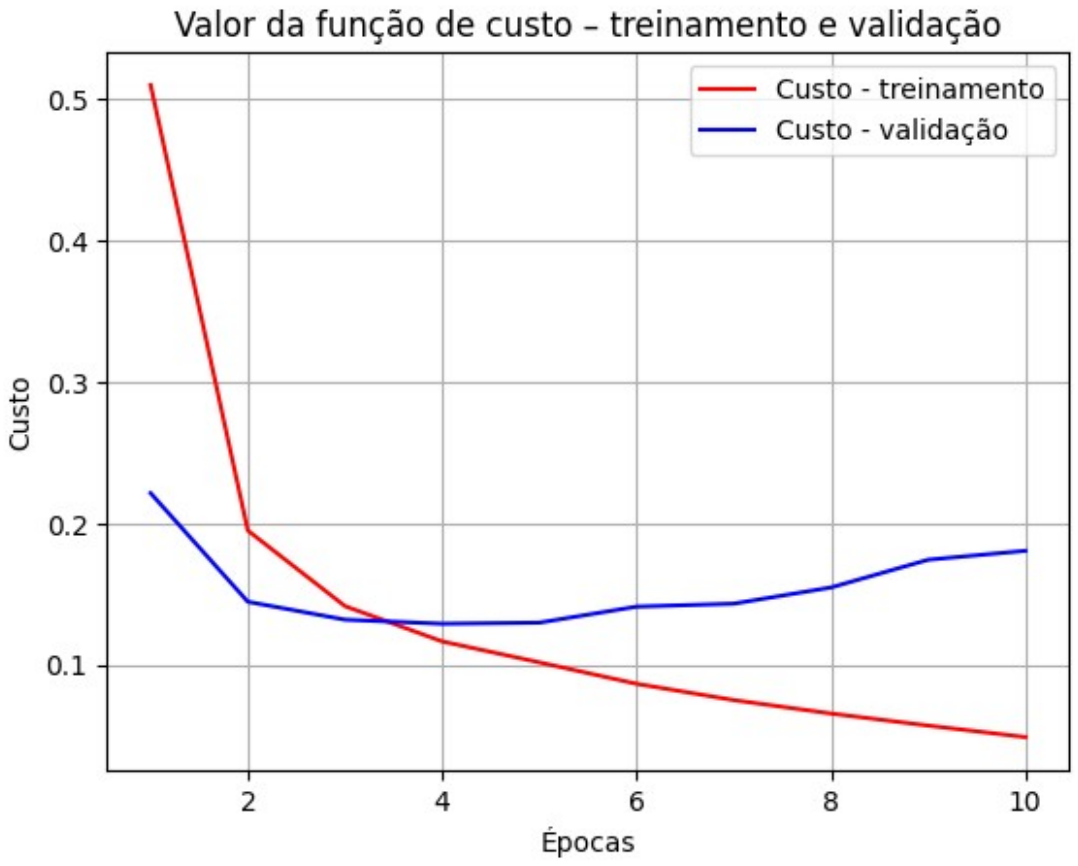
```
20/20 ━━━━━━━━━━━━━━━━━━━━ 944s 46s/step - accuracy: 0.9436 - loss:
0.1484 - val_accuracy: 0.9575 - val_loss: 0.1319
Epoch 4/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 951s 46s/step - accuracy: 0.9508 - loss:
0.1215 - val_accuracy: 0.9553 - val_loss: 0.1291
Epoch 5/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 954s 47s/step - accuracy: 0.9599 - loss:
0.1036 - val_accuracy: 0.9607 - val_loss: 0.1298
Epoch 6/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 943s 46s/step - accuracy: 0.9628 - loss:
0.0890 - val_accuracy: 0.9512 - val_loss: 0.1412
Epoch 7/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 928s 45s/step - accuracy: 0.9690 - loss:
0.0784 - val_accuracy: 0.9541 - val_loss: 0.1433
Epoch 8/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 935s 46s/step - accuracy: 0.9721 - loss:
0.0679 - val_accuracy: 0.9531 - val_loss: 0.1547
Epoch 9/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 943s 46s/step - accuracy: 0.9808 - loss:
0.0589 - val_accuracy: 0.9475 - val_loss: 0.1744
Epoch 10/10
20/20 ━━━━━━━━━━━━━━━━━━━━ 939s 46s/step - accuracy: 0.9828 - loss:
0.0503 - val_accuracy: 0.9478 - val_loss: 0.1807
```

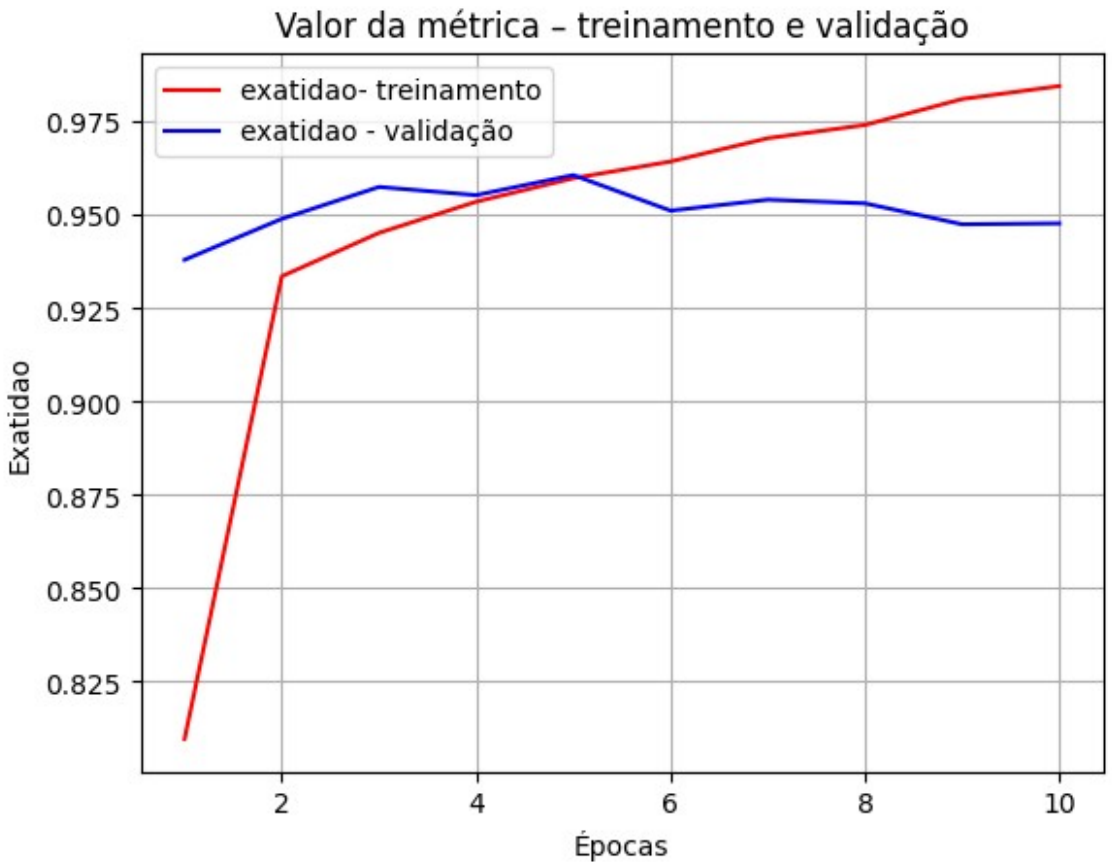
Importante:

Usar um Dataset com o método `repeat` gera infinitos exemplos de treinamento. Nesse caso, no treinamento deve-se usar o argumento `steps_per_epoch` e `validation_steps` para definir quantos lotes por época são desejados.

```
# Salva treinamento na variável history para visualização
resultado = results.history

# Salva custos, métricas e epocas em vetores
custo = resultado['loss']
acc = resultado['accuracy']
val_custo = resultado['val_loss']
val_acc = resultado['val_accuracy']

# Cria vetor de épocas
epocas = range(1, len(custo) + 1)

# Gráfico dos valores de custo
plt.plot(epocas, custo, 'r', label='Custo - treinamento')
plt.plot(epocas, val_custo, 'b', label='Custo - validação')
plt.title('Valor da função de custo – treinamento e validação')
plt.xlabel('Épocas')
plt.ylabel('Custo')
plt.legend()
```

```
plt.grid()
plt.show()

# Gráfico dos valores da métrica
plt.plot(epocas, acc, 'r', label='exatidao- treinamento')
plt.plot(epocas, val_acc, 'b', label='exatidao - validação')
plt.title('Valor da métrica – treinamento e validação')
plt.xlabel('Épocas')
plt.ylabel('Exatidao')
plt.grid()
plt.legend()
plt.show()
```





5.7 Avaliação e teste da RNA

Vamos avaliar o desemepnho da RNA usando o método `evaluate`com os dados de teste.

```
# Calcula função de custo e exatidão para os dados de teste
loss, accuracy = rna.evaluate(balanced_ds_test, steps=10)

print(' ')
print("Função de custo:", loss)
print("Exatidão:", accuracy)
```

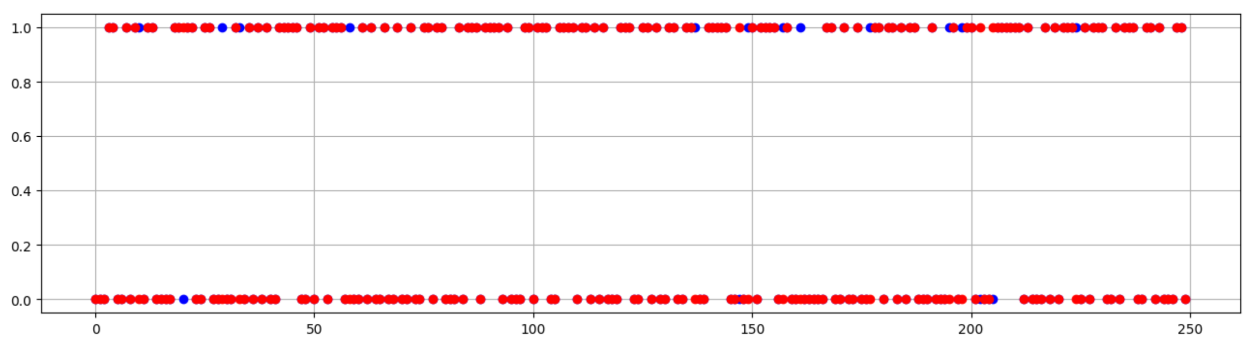
```
10/10 ━━━━━━━━━━━━━━━━━━━━ 1457s 146s/step - accuracy: 0.9487 - loss:
0.1833

Função de custo: 0.18400561809539795
Exatidão: 0.948168933391571
```

Realizar previsões

Vamos verificar os resultados da RNA com detalhe usando o método predict e depois verificar o acerto para cada uma das classes.

```python
# Gera lote de dados
x, y_real = next(iter(balanced_ds_test))

# Calcula previsão da RNA para um lote de imagens
y_prev = rna.predict(x)
classe_prev = np.round(y_prev)

# Redimensiona vetor de classes reais
classe_real = np.reshape(y_real, (batch_size, 1))

# Exatidão obtida
print('Exatidão =', 1.0 - np.mean(np.abs(classe_real-classe_prev)))

# Mostra primeiro exemplo do lote
plt.figure(figsize=(16,4))
plt.plot(y_real[:250], 'bo')
plt.plot(classe_prev[:250], 'ro')
plt.grid()
plt.show()
```

```
128/128 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Exatidão = 0.949951171875
```




```python
rna.save('rna.keras')

# Carrega modelo salvo
rna2 = tf.keras.models.load_model('rna.keras')
rna2.summary()

# Calcula função de custo e exatidão para os dados de teste
loss, accuracy = rna.evaluate(test_ds, steps=10, verbose=0)

print(' ')
print("Função de custo:", loss)
print("Exatidão:", accuracy)
```

```
Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
━━━━━━━━━━━━━━━━┛
```

```
┃ Layer (type)                         ┃ Output Shape                ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇
━━━━━━━━━━━━━━━━━┩
│ dense (Dense)                        │ (None, 128)                 │
3,840 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ dense_1 (Dense)                      │ (None, 64)                  │
8,256 │
├──────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ dense_2 (Dense)                      │ (None, 1)                   │
65 │
└──────────────────────────────────────┴─────────────────────────────┴
─────────────────┘

 Total params: 36,485 (142.52 KB)

 Trainable params: 12,161 (47.50 KB)

 Non-trainable params: 0 (0.00 B)

 Optimizer params: 24,324 (95.02 KB)


Função de custo: 0.04559134691953659
Exatidão: 0.988207995891571
```