Aula 8

"Data Pipelines" - Parte 1

Extração e Transformação dos dados

Eduardo Lobo Lustosa Cabral

1. Objetivos

Apresentar o conceito de "data pipelines".

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

Apresentar as etapas de extração e transformação de um "data pipeline".

Apresentar exemplos das etapas de extração e transfromação de um "pipelines" para diferentes tipos de dados.

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

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 podem ser usados por uma RNA;
- **Carregar** → alimenta a RNA com dados durante o seu treinamento ou para realizar previsões quando colocada em operação.

Nessa aula veremos as etapas de Extração e Transformação de um Pipeline.

Cada tipo de dado exige um "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 3 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).

Por exemplo, um pipeline para um problema de processamento de imagens pode agregar dados de vários arquivos, aplicar transformações aleatórias em cada imagem ("data augmentation") e juntar imagens selecionadas aleatoriamente em um lote para treinamento.

O pipeline para um problema de processamento de texto pode envolver a extração de símbolos de dados de textos, convertê-los em vetores "embedding" e juntar sequências de comprimentos diferentes em lotes para treinamento.

3. Mecânica básica das etapas de extração e transformação

O módulo `tf.data` possui a classe `tf.data.Dataset`, que armazena um conjunto de dados.

Um objeto `Dataset` representa uma sequência de elementos, na qual cada elemento consiste em um ou mais componentes. Por exemplo, em um pipeline de imagem, um elemento pode ser um único exemplo de treinamento, com um par de tensores que representam a imagem e seu rótulo.

Para criar um pipeline de entrada, deve-se começar com um conjunto de dados e criar um objeto **`Dataset`**. Um `Dataset` é um objeto iterável do Python. Isso torna possível utilizar seus elementos usando, por exemplo, um loop `for`.

Tendo um objeto `Dataset`, é possível transformá-lo em um novo `Dataset` usando diversos métodos. Por exemplo, pode-se aplicar transformações em cada elemento com os métodos `map()`, ou em vários elementos com o método `batch()`. A lista completa de métodos disponíveis para os objetos `Dataset` pode ser vista na documentação https://www.tensorflow.org/api_docs/python/tf/data/Dataset.

3.1 Criar um `Dataset`

Existem várias maneiras distintas de criar um conjunto de dados (`Dataset`):

- Inserindo dados;
- A partir de uma fonte de dados armazenada em memória ou a partir de um ou mais arquivos;
- A partir de transformações de dados de um ou mais objetos `Dataset`.

Um objeto `Dataset` do TensorFlow é um gerador de dados.

Os códigos das células baixo mostram exemplos de como criar objetos `Dataset`.

Para criar um `Dataset` simples com números inteiros, tem-se:

```
# Cria um objeto dataset
dataset = tf.data.Dataset.from_tensor_slices([0., 1., 2., 3., 4., 5.,
6., 7., 8., 9., 10.])
dataset
```

```
<_TensorSliceDataset element_spec=TensorSpec(shape=(),
dtype=tf.float32, name=None)>
```

- O método `tf.data.Dataset.from_tensor_slices()` transforma um tensor em um `Dataset`.

Acessar elementos de um `Dataset`

Pode-se acessar os elementos de um `Dataset` de várias formas diferentes.

Uma forma simples é iterando nos elementos do `Dataset`.

```
# Itera nos elementos do dataset e mostra cada um
for elemento in dataset:
    print(elemento.numpy())
```

```
0.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0
```

Pode-se também acessar os elementos de um objeto `Dataset` criando explicitamente um iterador Python, usando a classe `iter`, e carregar seus elementos usando a função `next`.

O método `next()` carrega o próximo elemento do dataset usando o iterador `iter`.

```
it = iter(dataset)

print(next(it))
print(next(it).numpy())
print(next(it).numpy())
for _ in range(10):
  try:
    print(next(it).numpy())
  except StopIteration:
    break
  print(next(it).numpy())
```

```
tf.Tensor(0.0, shape=(), dtype=float32)
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0
```

Outra forma é usar o método `take`.

```
# Itera no dataset para pegar exemplos
for num in dataset.take(6):
    print('Número =', format(num))
```

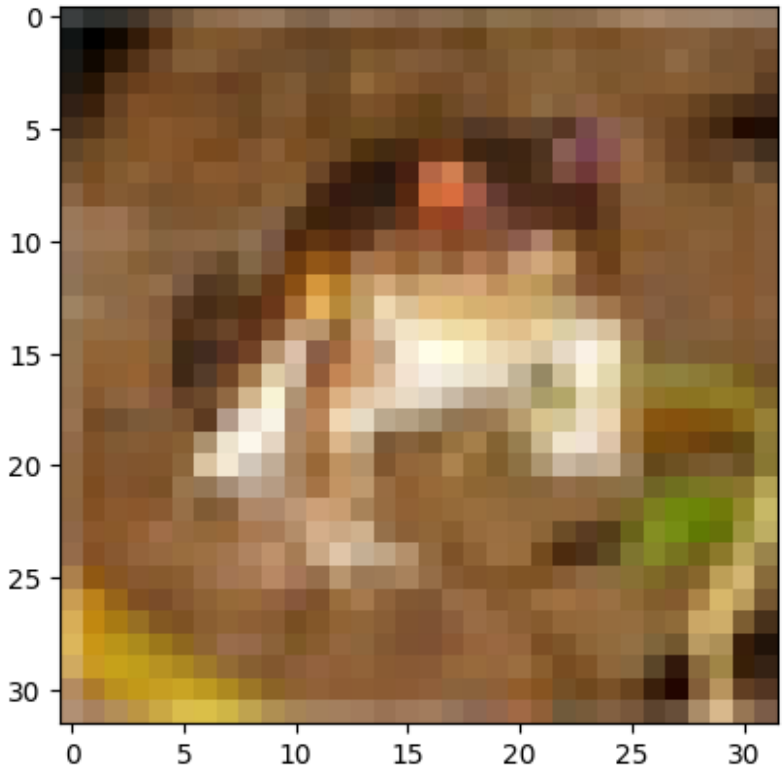
```
Número = 0.0
Número = 1.0
Número = 2.0
Número = 3.0
Número = 4.0
Número = 5.0
```

Criar um `Dataset` com dados já em tensores Numpy

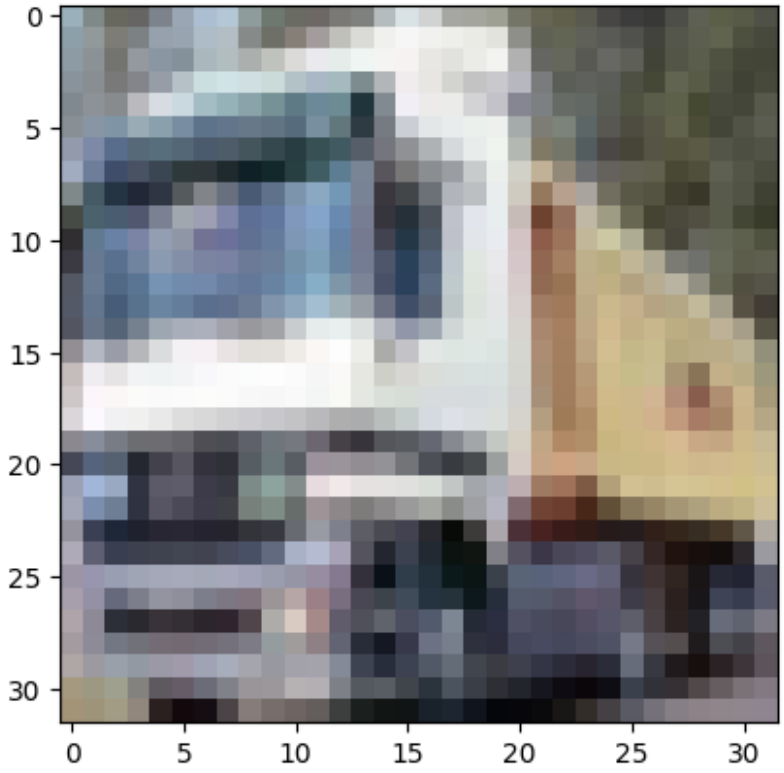
Para transformar dados que já estão em tensores Numpy em um `Dataset`, usa-se o método `tf.data.Dataset.from_tensor_slices()` da seguinte forma:

```
# Extrai conjunto de dados Cifar10 da coleção do Keras
(x_train, y_train), (x_val, y_val) =
tf.keras.datasets.cifar10.load_data()

# Dimensão dos dados
print('Dados de treinamento:', x_train.shape, y_train.shape)
print('Dados de validação:', x_val.shape, y_val.shape)

# Transforma tensores Numpy em Dataset
data_ds = tf.data.Dataset.from_tensor_slices((x_train, y_train))

# Mostra alguns exemplos do Dataset
for image, label in data_ds.take(5):
    print("Classe: {}".format(label))
    plt.imshow(image)
    plt.show()
```

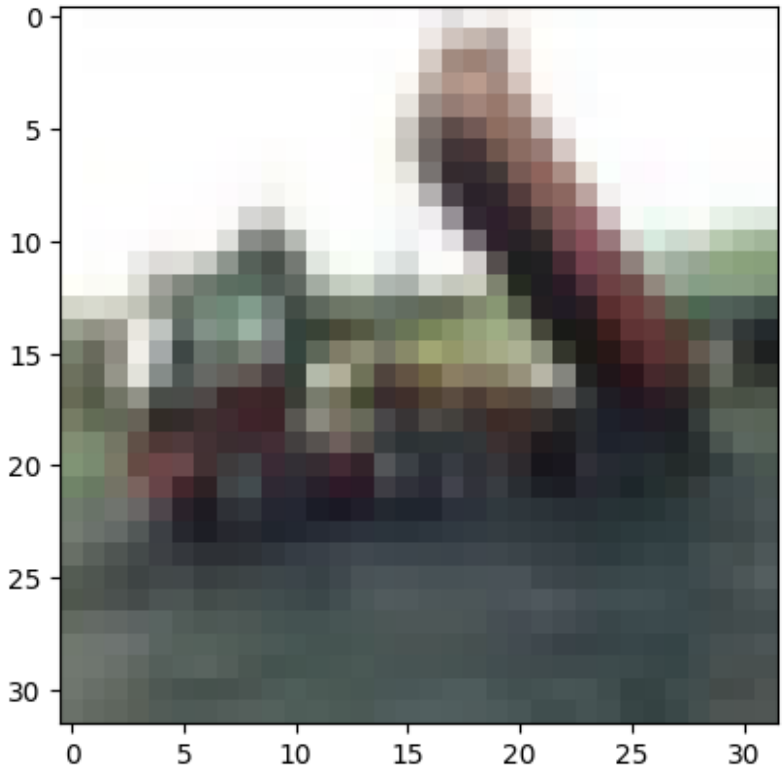
```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-
python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 4s 0us/step
Dados de treinamento: (50000, 32, 32, 3) (50000, 1)
Dados de validação: (10000, 32, 32, 3) (10000, 1)
Classe: [6]
```



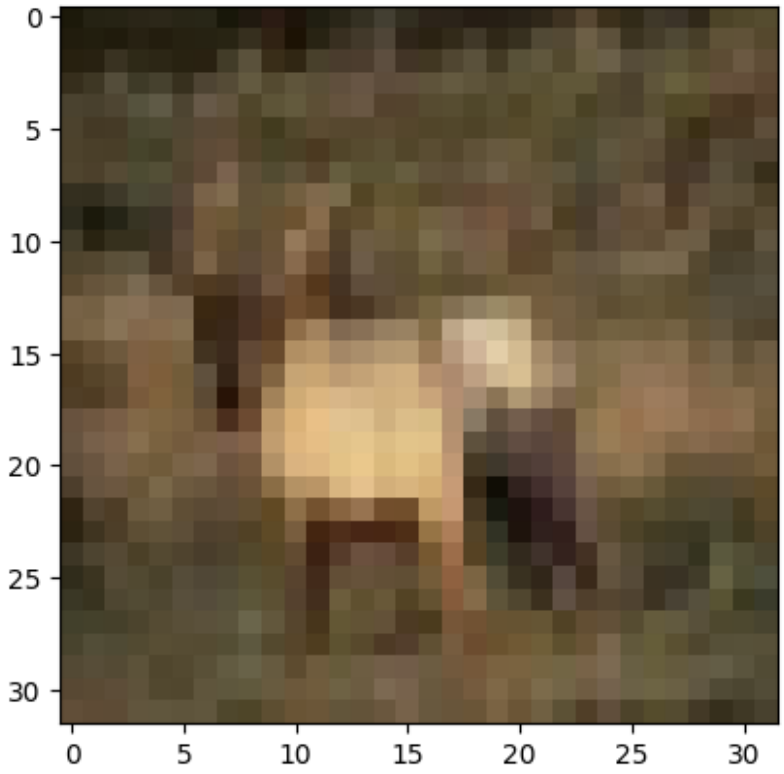

```
Classe: [9]
```



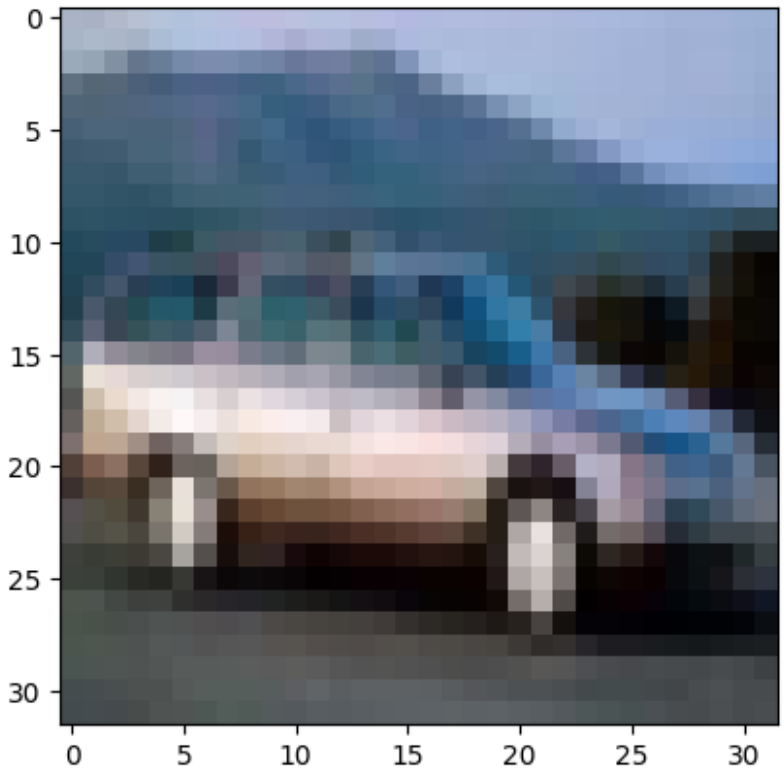
```
Classe: [9]
```




```
Classe: [4]
```




```
Classe: [1]
```



3.2 Estrutura dos objetos `Dataset`

Um `Dataset` contém componentes dispostos em uma estrutura, sendo que os componentes podem ser de tipos diferentes. O tipo de elemento de cada componente pode ser: `Tensor`, `SparseTensor`, `RaggedTensor`, `TensorArray` ou mesmo um `Dataset`.

A propriedade `element_spec` permite inspecionar o tipo de cada elemento dos componentes de um `Dataset`. A propriedade retorna a estrutura dos elementos do `Dataset`.

Seguem alguns exemplos.

```
# Cria dataset com um tensor de dimensão 4x10
dataset1 = tf.data.Dataset.from_tensor_slices(tf.random.uniform([4,
10]))

# Verifica tipo dos elementos
dataset1.element_spec
```

```
TensorSpec(shape=(10,), dtype=tf.float32, name=None)
```

- Esse `Dataset`contém 4 linhas de dados, sendo cada linha um elemento de dimensão 1x10.

```
# Cria dataset com dois tensores
dataset2 = tf.data.Dataset.from_tensor_slices(
    (tf.random.uniform([4]),
    tf.random.uniform([4, 100], maxval=100, dtype=tf.int32)))

# Verifica tipo dos elementos
dataset2.element_spec
```

```
(TensorSpec(shape=(), dtype=tf.float32, name=None),
 TensorSpec(shape=(100,), dtype=tf.int32, name=None))
```

- Esse `Dataset` possui dois componentes. O primeiro possui elementos escalar e o segundo possui elementos com dimensão 1x100.

Pode-se criar um novo `Dataset` a partir de outros datasets usando o método `zip`.

```
# Cria um novo dataset com o dataset1 e dataset2
dataset3 = tf.data.Dataset.zip((dataset1, dataset2))

# Verifica tipo dos elementos
dataset3.element_spec
```

```
(TensorSpec(shape=(10,), dtype=tf.float32, name=None),
 (TensorSpec(shape=(), dtype=tf.float32, name=None),
  TensorSpec(shape=(100,), dtype=tf.int32, name=None)))
```

- O método `zip` une dois ou mais `Datasets`.

```
for elem in dataset3.take(2):
    print(elem,'\n')
```

```
(<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([0.0086, 0.3543, 0.6571, 0.4238, 0.7526, 0.8052, 0.057 , 0.9048,
       0.9276, 0.1276], dtype=float32)>, (<tf.Tensor: shape=(),
dtype=float32, numpy=0.40695214>, <tf.Tensor: shape=(100,),
dtype=int32, numpy=
array([10, 47, 44, 63, 29, 45, 23, 41, 66, 52, 10, 94, 85, 35, 65, 38,
98,
       98,  8, 26, 20, 20, 77,  1, 79, 28, 74, 64, 55, 66, 43, 25,  4,
5,
       90, 27, 90, 15, 13, 38, 61, 96, 28, 89, 91,  8, 41, 57, 88, 23,
31,
        2, 33, 63, 31, 58, 14, 45, 79, 33, 27,  6, 88, 21, 87, 83, 36,
73,
       31, 28, 67, 77, 53,  3, 59, 91,  6, 10, 13, 59, 62, 15, 55, 82,
33,
       15, 15, 34, 65, 32, 98, 66, 69, 64, 89, 32, 90, 75, 12, 36],
      dtype=int32)>))

(<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([0.0747, 0.7867, 0.8103, 0.7011, 0.9135, 0.8189, 0.1994, 0.5262,
       0.4315, 0.4073], dtype=float32)>, (<tf.Tensor: shape=(),
dtype=float32, numpy=0.5044131>, <tf.Tensor: shape=(100,),
dtype=int32, numpy=
array([90, 66, 98, 75, 92, 39, 56, 99, 77, 84, 34, 58, 16, 75, 69, 55,
42,
       50,  9,  9, 31, 31, 40,  1,  2, 27, 37, 75, 46, 57, 95, 89, 38,
75,
       31, 29, 20, 24, 72, 54, 62, 55, 62,  5, 31,  0, 58, 81, 47, 19,
61,
       11, 34,  2, 95, 81, 89, 13, 38, 34, 43, 75, 27, 90,  9, 17, 25,
6,
       77, 20, 32, 88, 69, 85, 99, 77,  1, 93, 83, 80, 36, 65, 58, 66,
50,
       76, 88, 70, 51, 77, 97, 29, 60, 14, 72, 43, 35, 49,  2, 62],
      dtype=int32)>))
```

3.2 Transformação de dados

Os dados de um `Dataset` podem ser transformados usando os métodos `map`, `apply` e `filter`. O método `map` aplica uma determinada função em cada elemento do `Dataset` e os métodos `apply` e `filter` aplicam uma função em todo o `Dataset`.

Para usar esses métodos tem-se que criar uma função para ser aplicada pelos mesmos.

O código abaixo apresenta um exemplo de uso do método `map` para transformar um `Dataset` com números inteiros em reais e depois calcular a raiz quadrada. O primeiro passo é criar um `Dataset`.

```
# Cria dataset com números aleatórios inteiros
dataset = tf.data.Dataset.from_tensor_slices(tf.random.uniform([4,
10], minval=1, maxval=10, dtype=tf.int32))

# Apresenta dataset
for x in dataset:
    print(x.numpy())
```

```
[9 9 7 2 4 3 4 2 3 7]
[1 9 1 4 3 6 4 4 7 1]
[1 7 6 9 9 7 2 7 7 8]
[4 9 8 3 3 4 4 9 6 5]
```

```
## Define função para transformar números inteiros em reais e calcular
a raiz quadrada
def raiz(x):
    y = tf.math.sqrt(tf.cast(x, tf.float32))
    return y

# Aplica função raiz
dataset = dataset.map(raiz)

# Apresenta resultados
print(list(dataset.as_numpy_iterator()),'\n')

# Apresenta dataset
for x in dataset:
    print(x.numpy())
```

```
[array([1.7321, 1.7321, 1.6266, 1.1892, 1.4142, 1.3161, 1.4142,
1.1892,
       1.3161, 1.6266], dtype=float32), array([1.    , 1.7321, 1.    ,
1.4142, 1.3161, 1.5651, 1.4142, 1.4142,
       1.6266, 1.    ], dtype=float32), array([1.    , 1.6266, 1.5651,
1.7321, 1.7321, 1.6266, 1.1892, 1.6266,
       1.6266, 1.6818], dtype=float32), array([1.4142, 1.7321, 1.6818,
1.3161, 1.3161, 1.4142, 1.4142, 1.7321,
       1.5651, 1.4953], dtype=float32)]

[1.7321 1.7321 1.6266 1.1892 1.4142 1.3161 1.4142 1.1892 1.3161
1.6266]
[1.     1.7321 1.     1.4142 1.3161 1.5651 1.4142 1.4142 1.6266
1.    ]
[1.     1.6266 1.5651 1.7321 1.7321 1.6266 1.1892 1.6266 1.6266
1.6818]
[1.4142 1.7321 1.6818 1.3161 1.3161 1.4142 1.4142 1.7321 1.5651
1.4953]
```

- A função `cast()` do TensorFlow serve para alterar tipo de dado. No caso foi feita uma alteração de inteiro (`int32`) pata real (`float32`).
- O método `as_numpy_iterator()` do Numpy serve para iterar em um conjunto de tensores.

Mais detalhes de como usar os métodos `Dataset.apply()` e `Dataset.filter()` podem ser obtidos em https://www.tensorflow.org/api_docs/python/tf/data/Dataset

4. Carregar dados em um `Dataset` (Etapa de extração)

Dados podem ser de vários formatos e estarem em diversas formas.

O módulo `tf.data` do TensorFlow fornece métodos para carregar praticamente qualquer tipo de dado.

4.1 Tensores NumPy

Se o conjunto de dados for pequeno e couber na memória do computador, então, a forma mais simples para criar um `Dataset` é converter os dados em tensores e carregá-los com o método `from_tensor_slices()`.

Por exemplo, o conjunto de dados de dígitos MINIST pode ser importado da coleção de dados do Keras e transformado em um `Dataset` com o seguinte código.

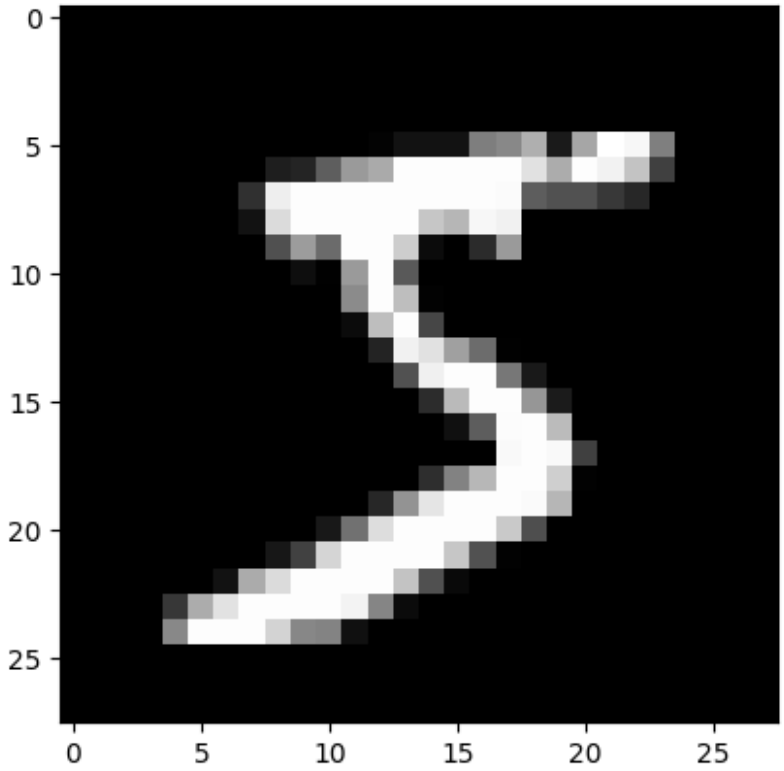
```
# Importa dados MNIST do Keras
train, test = tf.keras.datasets.mnist.load_data()

# Separa imagens e rótulos
images_train, labels_train = train

# Normaliza pixels da imagem
images_train = images_train/255.

# Cria dataset
dataset = tf.data.Dataset.from_tensor_slices((images_train,
labels_train))
dataset
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step

<_TensorSliceDataset element_spec=(TensorSpec(shape=(28, 28),
dtype=tf.float64, name=None), TensorSpec(shape=(), dtype=tf.uint8,
name=None))>
```

Para visualizar um exemplo do `Dataset` pode-se usar o código a seguir.

```
# Obtém um elemento do Dataset
it = iter(dataset)
image, label = next(it)

print('Dimensão das imagens = ', image.shape)
print('Classe =', label.numpy())
plt.imshow(image, cmap='gray')
plt.show()
```

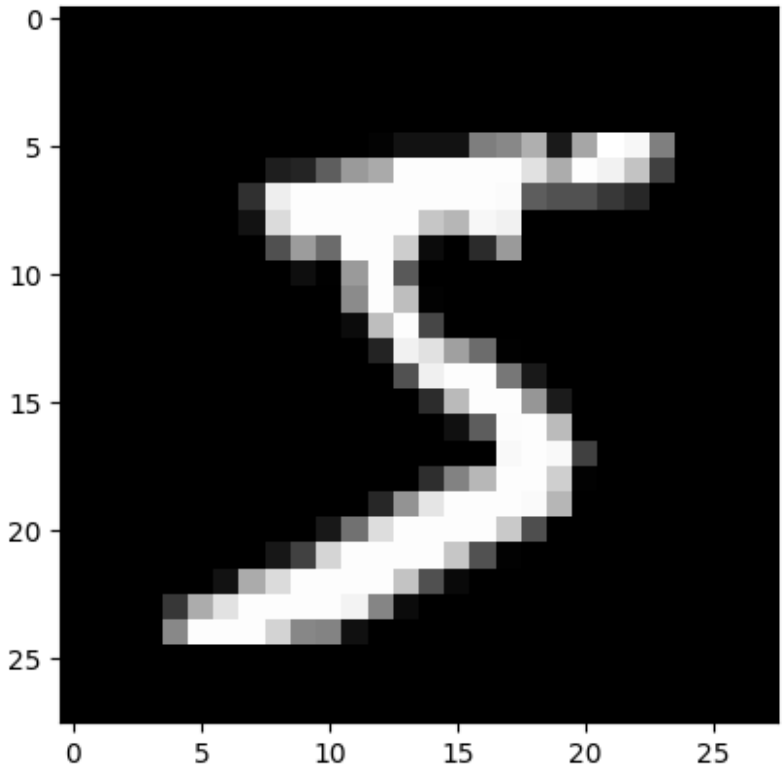
```
Dimensão das imagens =  (28, 28)
Classe = 5
```



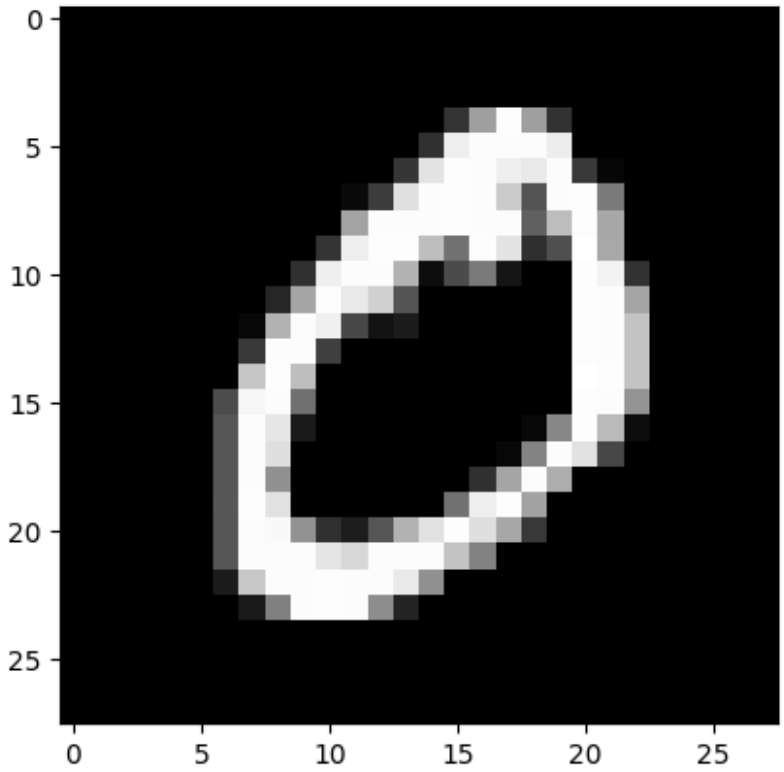

```
def pre_proc(img, label):
    img = tf.cast(img, tf.float32)
    img = img/255.
    label_hot = tf.one_hot(label, 10)
    return img, label_hot

# Importa dados MNIST do Keras
train, test = tf.keras.datasets.mnist.load_data()

# cria dataset
dataset = tf.data.Dataset.from_tensor_slices(train)

for data in dataset.take(1):
```

```
    img, label = data
    print(img.shape, label)

# Cria dataset
dataset_pre = dataset.map(pre_proc)

for image, label in dataset_pre.take(2):
    print('Dimensão das imagens = ', image.shape)
    print('Classe =', label.numpy())
    plt.imshow(image, cmap='gray')
    plt.show()
    print('Valores mínimos e máximos:', np.min(image), np.max(image))
```

```
(28, 28) tf.Tensor(5, shape=(), dtype=uint8)
Dimensão das imagens =  (28, 28)
Classe = [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```




```
Valores mínimos e máximos: 0.0 1.0
Dimensão das imagens =  (28, 28)
Classe = [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```




```
Valores mínimos e máximos: 0.0 1.0
```

4.2 Gerador de dados

Outra forma muito utilizada para carregar dados é usar um gerador de dados.

Um gerador de dados é usado quando o conjunto de dados é muito grande e não cabe na memória, assim, tem-se que carregar os dados lote a lote.

Geradores de dados são muito usados para imagens e o seu uso permite também realizar "data augmentation".

No código abaixo é apresentado um gerador de dados que toda vez que é chamado retorna (`yield`) o próximo número de um sequencia de 1 a infinito.

```
# Define gerador de dados
def seq(stop):
    i = 0
    while i<stop:
        yield i
        i += 1

# Chama gerador de dados 5 vezes
for n in seq(10):
    print(n)
```

```
0
1
2
3
4
5
6
7
8
9
```

```
sq = seq(10)
n1 = next(sq)
n2 = next(sq)
n3 = next(sq)
print(n1, n2, n3)
```

```
0 1 2
```

- Observe que um gerador de dados é diferente de uma função, pois ele retorna sempre o próximo elemento da sequência, ou seja, um gerador de dados "possui" memória.

O método `from_generator` converte um gerador de dados Python em um `Dataset`. Esse método recebe o gerador como entrada.

No código abaixo é apresentado um exemplo de construção de um `Dataset` usando o gerador `seq` definido anteriormente.

```
ds_counter = tf.data.Dataset.from_generator(seq, args=[25],
output_types=tf.int32, output_shapes = ())
```

- Argumento `args` serve para passar valores para o gerador de dados. Nesse caso, esse argumento representa a variável `stop` do gerador `seq`.
- Argumento `output_types` é obrigatório e define o tipo de dado do `Dataset`.
- Argumento `output_shapes` é opcional e nesse caso é deixado como sendo indefinido para poder retornar sequências de comprimentos diferentes.

O uso de gerador para obter dados pode ser realizado com o método `repeat()` de acordo com o código abaixo.

```
ds_counter = ds_counter.repeat().batch(10)

for count_batch in ds_counter.take(10):
    print(count_batch.numpy())
```

```
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
```

```
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]
 [ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]]
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]
 [ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]]
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]
 [ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]]
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]
 [ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]]
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]
 [ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]]
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
```

```
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]
 [ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]]
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]
 [ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]]
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]
 [ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]]
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]
 [ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]]
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]
 [ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24  0  1  2  3  4]
 [ 5  6  7  8  9 10 11 12 13 14]
 [15 16 17 18 19 20 21 22 23 24]]
```

- O método `batch` define o número de elementos a ser gerado em cada lote de dados, ou seja, o número de elementos por lote que será usado no treinamento.
- O método `take` define o número de lotes que se deseja acessar.
- O método `repeat` permite repetir os elementos do gerador.

Outro exemplo de um gerador mais completo, que retorna uma tuple de dados, que representa, por exemplo, dados de entrada e das saídas desajadas, é definido no código abaixo.

Nesse gerador o primeiro valor é um escalar e o segundo é um tensor de dimensão (1, 8)

```
# Define gerador de dados
def gen_series():
    i = 0
    while True:
        # Gera vetor de números aleatórios
        yield i, np.random.normal(size=(1,8))
        i += 1

# Chama o gerador 5 vezes
for i, series in gen_series():
    print(i, ":", str(series))
    if i > 5:
        break
```

```
0 : [[ 3.1366 -0.9517  0.1932  1.0341 -0.2889 -0.3345 -0.0511 -
0.0253]]
1 : [[ 1.7037  0.521   0.7728 -0.6077 -0.0822 -0.1444  0.1334 -
0.7317]]
2 : [[ 1.4233 -0.0723  0.1342 -0.0999  1.1261 -1.1648 -0.6165 -
0.3857]]
3 : [[-1.3947 -0.7104  1.6269  1.2922 -0.5516  1.0242  1.3414 -
0.2203]]
4 : [[ 0.802   1.4782 -0.5709  1.2697  1.7159  0.0795  0.027  -
0.6421]]
5 : [[ 0.4145 -0.9917 -0.824  -0.5563  0.7255 -0.0713 -1.4946
1.5246]]
6 : [[ 1.3971  0.4086  1.593   1.5666  1.3607  1.4281 -0.4279 -
2.0083]]
```

- O gerador possui um loop infinito que somente é quebrado quando não é mais chamado.
- A primeira saída é um número escalar inteiro (`int32`) e a segunda é um vetor de dimensão (1, 8) com números reais (`float32`) aleatórios com distribuição gaussiana.

Para criar um `Dataset` com esse gerador usa-se o método `from_generator` como anteriormente.

Nesse caso é necessário definir os tipos de dados e as dimensões dos dados para as duas saídas do gerador.

```
# Cria Dataset com o gerador gen_series
ds_series = tf.data.Dataset.from_generator(
    gen_series,
    output_types=(tf.int32, tf.float32),
    output_shapes=((), (1,8,)))

# Apresenta carecterísticas gerais do dataset
ds_series.element_spec
```

```
(TensorSpec(shape=(), dtype=tf.int32, name=None),
 TensorSpec(shape=(1, 8), dtype=tf.float32, name=None))
```

Esse `Dataset` é usado da mesma forma que um `Dataset` regular.

No código abaixo o `Dataset` `ds_series` é inicializado para criar um lote com 10 exemplos (`batch(10)`) e os dados são embaralhados aleatóriamente usando o método `shuffle`.

```
# Inicializa dataset para gerar lote com 10 exemplos
ds_series_batch = ds_series.shuffle(20).batch(10)

# Iterage um única vez no dataset para obter um lote
ids, sequence_batch = next(iter(ds_series_batch))
print(ids.numpy())
print()
print(sequence_batch.numpy())
```

```
[ 4  5 19 22 20 10 12 17 16  9]

[[[-0.915  -0.6744  0.6045  0.7726  1.3728  0.7673  0.1264 -0.353 ]]

 [[ 1.4564  0.3418  0.5189 -0.9314 -0.7527  0.3148 -0.0191 -1.8737]]

 [[-0.0797  1.0483  3.5719  0.6543  1.0144  1.1172  0.3436 -1.257 ]]

 [[ 1.0898 -0.9431  3.5266 -0.6102  0.3006 -0.2294 -0.4411  0.6723]]

 [[ 1.5255 -0.8832  0.8075  1.1152  0.0393  1.8888  0.8293  0.0986]]

 [[-0.4972 -0.3698 -0.7024  0.0527 -0.1413 -1.1448  1.2562  0.8898]]

 [[-0.6256 -0.9004 -1.5259 -0.7015 -0.4382 -0.5694  1.0348 -0.1534]]

 [[ 1.2383  0.4813  0.3     0.0677 -0.4845  1.2825 -0.3913  0.402 ]]

 [[ 0.4482 -0.3397  0.7335  0.174  -0.8225  0.7078  0.4329 -1.321 ]]

 [[-0.7209  1.1144  0.8163 -0.2468 -0.3007 -0.9533  0.2007 -0.5427]]]
```

- No caso `shuffle(20)` representa que é formado um buffer de 20 exemplos de dados e o lote é sorteado aleatóriamente entre esses 20 exemplos. Observa-se que quanto maior o buffer, melhor é realizado o embaralhamento dos dados.

4.3 TFRecord

Quando o conjunto de dados for muito grande, uma opção para ler os dados de maneira eficiente é serializar os dados e armazená-los em um conjunto de arquivos binários com tamanho de, por exemplo, 100-200 MB cada, que podem ser lidos linearmente.

O formato TFRecord é um formato simples para armazenar dados em uma sequência de registros binários.

A classe `tf.data.TFRecordDataset` permite que os dados sejam carregados por meio de "streaming" a partir de um ou mais arquivos do tipo TFRecord e que sejam usados durante o treinamento de uma RNA em tempo real.

Essa forma de armazenar dados é extremamente útil se os dados estiverem sendo transmitidos por uma rede.

Observa-se que existem muitos conjuntos de dados nesse formato.

Não veremos detalhes desse tipo de dado, assim, para obter mais informações sobre TFRecord ver Loading TFRecords.

Segue um exemplo simples de código que mostra como usar esse tipo de formato de dados. Nesse exemplo, são usados os dados de teste do conjunto de dados de nomes de ruas da França (FSNS).

```
# Define arquivos com os dados (são usados 2 arquivos do conjunto)
fsns_test_file = tf.keras.utils.get_file("fsns.tfrec",

"https://storage.googleapis.com/download.tensorflow.org/data/fsns-
20160927/testdata/fsns-00000-of-00001")

# Apresenta nome base do subdiretório onde estão os arquivos
print(fsns_test_file)
```

```
Downloading data from
https://storage.googleapis.com/download.tensorflow.org/data/fsns-
20160927/testdata/fsns-00000-of-00001
7904079/7904079 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
/root/.keras/datasets/fsns.tfrec
```

- A função `tf.keras.utils.get_file()` carrega um ou mais arquivos da URL fornecida e coloca-os no diretório (ou arquivo) "~./Keras/datsets/nome", onde "nome" é o nome do subdiretório (ou arquivo).
- "fsns.tfrec" é o nome do subdiretório onde vão ser colocados os arquivos lidos e o outro argumento é a URL onde se encontram os arquivos de origem. Essa função também pode descompactar arquivos ao carregá-los. Existem muitos argumentos

que podem ser definidos para essa função, para obter mais detalhes ver https://www.tensorflow.org/api_docs/python/tf/keras/utils/get_file.

- Observe que os arquivos já estão no formato TFRecords.

Para criar um `Dataset` com os dados contidos nesses arquivos, os nomes deles são passados pelo argumento `filenames` na inicialização do objeto `TFRecordDataset`. O argumento `filenames` pode ser uma string ou uma lista de strings.

O código a seguir apresenta como criar um `Dataset` usando os arquivos definidos na variável `fsns_test_file`.

```
dataset = tf.data.TFRecordDataset(filenames = [fsns_test_file])
dataset.element_spec
```

```
TensorSpec(shape=(), dtype=tf.string, name=None)
```

Para os dados poderem ser usados, os registros nos arquivos com os TFRecords precisam ser decodificados. O código a seguir realiza essa decodificação.

```
# Pega primeiro exemplo de dado
raw_example = next(iter(dataset))

# Decodifica dado
parsed = tf.train.Example.FromString(raw_example.numpy())

# Apresenta um exemplo de dado
parsed.features.feature['image/text']
```

```
bytes_list {
  value: "Rue Perreyon"
}
```

- O método `tf.train.Example.FromString()` realiza a transformação dos registros binários em string.
- Para mais detalhes sobre TFRecods ver https://www.tensorflow.org/tutorials/load_data/tfrecord.

4.4 Texto

Muitos conjuntos de dados de texto são distribuídos em vários arquivos. A classe `tf.data.TextLineDataset` fornece uma forma fácil de extrair linhas de texto de um ou mais arquivos.

Dados os nomes dos arquivos em uma lista de strings, a classe `TextLineDataset` produz uma string para cada linha de texto desses arquivos.

Para mais detalhes ver https://www.tensorflow.org/tutorials/load_data/text.

```
# Define diretório onde estão os arquivos
directory_url =
'https://storage.googleapis.com/download.tensorflow.org/data/illiad/'

# Define nomes dos arquivos
file_names = ['cowper.txt', 'derby.txt', 'butler.txt']

# Cria os caminhos completos para todos os arquivos
file_paths = [tf.keras.utils.get_file(file_name, directory_url +
file_name)
              for file_name in file_names]

# Aperesenta os caminhos
print(file_paths)
```

```
Downloading data from
https://storage.googleapis.com/download.tensorflow.org/data/illiad/
cowper.txt
815980/815980 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from
https://storage.googleapis.com/download.tensorflow.org/data/illiad/
derby.txt
809730/809730 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from
https://storage.googleapis.com/download.tensorflow.org/data/illiad/
butler.txt
807992/807992 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
['/root/.keras/datasets/cowper.txt',
'/root/.keras/datasets/derby.txt', '/root/.keras/datasets/butler.txt']
```

Cria `Dataset` com as linhas de texto dos arquivos e apresenta as 10 primeiras linhas dos primeiro arquivo.

```
# Cria dataset com linhas de texto
dataset = tf.data.TextLineDataset(file_paths)

# Apresenta 10 primeiras linhas
for line in dataset.take(10):
    print(line.numpy())
```

```
b"\xef\xbb\xbfAchilles sing, O Goddess! Peleus' son;"
b'His wrath pernicious, who ten thousand woes'
b"Caused to Achaia's host, sent many a soul"
b'Illustrious into Ades premature,'
b'And Heroes gave (so stood the will of Jove)'
b'To dogs and to all ravening fowls a prey,'
b'When fierce dispute had separated once'
b'The noble Chief Achilles from the son'
b'Of Atreus, Agamemnon, King of men.'
b"Who them to strife impell'd? What power divine?"
```

```
len(list(dataset))
```

```
49608
```

A classe `TextLineDataset` retorna todas as linhas dos arquivos, o que pode não ser desejado, por exemplo, se a a primeira linha for um cabeçalho, ou conter comentários.

Ao carregar um arquivo de texto, pode-se usar os métodos `skip()` ou `filter()`. Com esses métodos pode-se pular a primeira linha, ou quantas linhas for desejado e pegar as outras restantes.

Um exemplo de fazer essa operação segue abaixo com o conjunto de dados de texto criado anteriormente. Nesse exemplo pulam-se as 2 primeiras linhas.

```
# Cria novo dataset sem as duas primeiras linhas
linhas = dataset.skip(2)

# Apresenta 10 primeiras linhas do novo dataset
for line in linhas.take(10):
    print(line.numpy())
```

```
b"Caused to Achaia's host, sent many a soul"
b'Illustrious into Ades premature,'
b'And Heroes gave (so stood the will of Jove)'
b'To dogs and to all ravening fowls a prey,'
b'When fierce dispute had separated once'
b'The noble Chief Achilles from the son'
b'Of Atreus, Agamemnon, King of men.'
b"Who them to strife impell'd? What power divine?"
b"Latona's son and Jove's. For he, incensed"
b'Against the King, a foul contagion raised'
```

4.5 Arquivos tipo CSV

Arquivos no formato CSV podem ser carregados usando o Pandas e depois transformados em um objeto `Dataset`. Ou pode-se também carregá-los diretamente como um objeto `Dataset`.

Carregando arquivo com o Pandas

Se os dados couberem na memória, carregá-los com o Pandas é muito simples.

Como exemplo vamos usar o conjunto de dados "Titanic".

```
# Define nome do arquivo e URL com o arquivo de dados
titanic_file = tf.keras.utils.get_file("train.csv",
"https://storage.googleapis.com/tf-datasets/titanic/train.csv")

# Carrega dados com o Pandas
df = pd.read_csv(titanic_file, index_col=None)
df.head()
```

```
Downloading data from
https://storage.googleapis.com/tf-datasets/titanic/train.csv
30874/30874 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```


```
{"summary":"{\n  \"name\": \"df\",\n  \"rows\": 627,\n  \"fields\": [\
n    {\n      \"column\": \"survived\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 0,\n        \"min\": 0,\n
\"max\": 1,\n        \"num_unique_values\": 2,\n        \"samples\":
[\n          1,\n          0\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"sex\",\n      \"properties\": {\n        \"dtype\":
\"category\",\n        \"num_unique_values\": 2,\n        \"samples\":
[\n          \"female\",\n          \"male\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"age\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 12.511817629565819,\n
\"min\": 0.75,\n        \"max\": 80.0,\n        \"num_unique_values\":
76,\n        \"samples\": [\n          28.0,\n          59.0\
n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"n_siblings_spouses\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 1,\n        \"min\": 0,\n
\"max\": 8,\n        \"num_unique_values\": 7,\n        \"samples\":
[\n          1,\n          0\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"parch\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 0,\n        \"min\": 0,\n
\"max\": 5,\n        \"num_unique_values\": 6,\n        \"samples\":
[\n          0,\n          1\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"fare\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 54.597730499456304,\n        \"min\":
0.0,\n        \"max\": 512.3292,\n        \"num_unique_values\": 216,\
n        \"samples\": [\n          25.9292,\n          5.0\
n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"class\",\n      \"properties\": {\n        \"dtype\": \"category\",\
n        \"num_unique_values\": 3,\n        \"samples\": [\n
\"Third\",\n          \"First\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"deck\",\n      \"properties\": {\n
\"dtype\": \"category\",\n        \"num_unique_values\": 8,\n
\"samples\": [\n          \"C\",\n          \"D\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"embark_town\",\n
\"properties\": {\n        \"dtype\": \"category\",\n
\"num_unique_values\": 4,\n        \"samples\": [\n
\"Cherbourg\",\n          \"unknown\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    },\n    {\n      \"column\": \"alone\",\n      \"properties\": {\
```

```
n        \"dtype\": \"category\",\n        \"num_unique_values\": 2,\n
\"samples\": [\n          \"y\",\n          \"n\"\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    }\n  ]\n}","type":"dataframe","variable_name":"df"}
```


Para transformar um `DataFrame` do Pandas em um objeto `Dataset` é usado o método `from_tensor_slices`, que é o mesmo usado quando os dados estão no formato de tensores Numpy.

```
# Cria dataset com o dataframe Pandas
titanic_ds = tf.data.Dataset.from_tensor_slices(dict(df))

# Apresenta as 2 primeiras linhas dos dados
for feature_batch in titanic_ds.take(2):
    for key, value in feature_batch.items():
        print("  {}: {}".format(key, value))
    print(' ')
```

```
  survived: 0
  sex: b'male'
  age: 22.0
  n_siblings_spouses: 1
  parch: 0
  fare: 7.25
  class: b'Third'
  deck: b'unknown'
  embark_town: b'Southampton'
  alone: b'n'

  survived: 1
  sex: b'female'
  age: 38.0
  n_siblings_spouses: 1
  parch: 0
  fare: 71.2833
  class: b'First'
  deck: b'C'
  embark_town: b'Cherbourg'
  alone: b'n'
```

Carregando arquivo diretamente com `tf.data`

Uma forma mais eficiente é carregar os dados diretamente com o módulo `tf.data`, que permite carregar os dados na medida em forem necessários. Isso é importante para conjunto de dados muito grande que não cabe na memória do computador.

O método `experimental.make_csv_dataset` é capaz de carregar partes de um arquivo tipo CSV. Esse método fornece suporte para selecionar e transformar colunas, gerar lotes de dados etc.

O código abaixo mostra um exemplo de como usar esse método para criar um `Dataset` a partir do arquivo de dados tipo CSV do conjunto de dados "Titanic".

```
# Define nome e URL com o arquivo de dados
titanic_file = tf.keras.utils.get_file("train.csv",
"https://storage.googleapis.com/tf-datasets/titanic/train.csv")

# Cria objeto Dataset a partir do arquivo de dados
titanic_batches_ds = tf.data.experimental.make_csv_dataset(
                  titanic_file, batch_size=4,
                  label_name="survived")
```

- Nesse exemplo foi escolhido um lote de 4 exemplos, ou seja, 4 linhas. Assim, somente 4 linhas são carregas e somente quando necessárias.
- A coluna com a saída desejada ("survived") já foi selecionada ao carregar os dados de forma a gerar automatiamente dados de entrada e de saída.

Como exemplo, o código abaixo apresenta um lote de 4 exemplos de dados do "titanic", carregado do Dataset criado.

```
# Gera um lote de dados do dataset
for feature_batch, label_batch in titanic_batches_ds.take(1):
    print("survived: {}".format(label_batch))
    print("features:")
    for key, value in feature_batch.items():
        print("  {}: {}".format(key, value))
```

```
survived: [1 1 0 0]
features:
  sex: [b'male' b'female' b'male' b'male']
  age: [20. 52.  1. 28.]
  n_siblings_spouses: [0 1 5 0]
  parch: [0 0 2 0]
  fare: [ 7.2292 78.2667 46.9     9.5   ]
  class: [b'Third' b'First' b'Third' b'Third']
  deck: [b'unknown' b'D' b'unknown' b'unknown']
  embark_town: [b'Cherbourg' b'Cherbourg' b'Southampton'
b'Southampton']
  alone: [b'y' b'n' b'n' b'y']
```

Ao carregar os dados, é possível selecionar somente as colunas desejadas, conforme mostra o código a seguir, onde somente 3 colunas são selecionadas ('class', 'fare', 'survived').

```
# Cria objeto dataset a partir do arquivo de dados
titanic_batches = tf.data.experimental.make_csv_dataset(
    titanic_file, batch_size=4,
    label_name="survived", select_columns=['class', 'fare',
'survived'])
```

```
# Gera um lote de dados do dataset
for feature_batch, label_batch in titanic_batches.take(1):
    print("survived: {}".format(label_batch))
    print("features:")
    for key, value in feature_batch.items():
        print("  {}: {}".format(key, value))
```

```
survived: [1 0 0 0]
features:
  fare: [10.5    20.2125  6.975   9.5   ]
  class: [b'Second' b'Third' b'Third' b'Third']
```

Importante:

Para poder usar o Dataset acima com uma RNA, todos os dados tem que serem transformados para números. Para isso, o tensorFlow fornece o módulo `tf.feature_columns`, que permite transformar colunas de um conjunto de dados estruturados → isso será visto nas próximas aulas.

4.6 Conjuntos de arquivos

Existem muitos conjuntos de dados compostos por vários arquivos, onde cada arquivo é um exemplo ou alguns exemplos.

Nesse caso pode-se carregar todos os arquivos e colocá-los em um diretório temporário.

```
# Define URL onde estão os arquivos e os carrega no diretório
~./keras/datasets/flower_photos
flowers_root = tf.keras.utils.get_file(
    'flower_photos',

'https://storage.googleapis.com/download.tensorflow.org/example_images
/flower_photos.tgz',
    untar=True)

# Define caminho completo para colocar os arquivos
flowers_root = pathlib.Path(flowers_root)

print(flowers_root)
```

```
Downloading data from
https://storage.googleapis.com/download.tensorflow.org/example_images/
flower_photos.tgz
228813984/228813984 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
/root/.keras/datasets/flower_photos
```

- Nesse exemplo a função `pathlib.Path(dir)` cria o caminho para todos os subdiretórios onde foram copiados os arquivos.

- O diretório onde são colocados os arquivos é um padrão do tf.keras.

O diretório "flower_photos" contém um subdiretório para cada tipo (classe) de flor.

Para obter os nomes dos subdiretórios usa-se o método `glob()` → o método `glob()` obtém os nomes dos subdiretórios e dos arquivos dentro de um diretório.

Nesse exemplo, usa-se o argumento `"*"` no método `glob()` para selecionar tudo o que existe no diretório "flower_photos".

```
# Lista nomes dos subdireórios
for item in flowers_root.glob("*"):
    print(item.name)
```

```
sunflowers
daisy
dandelion
LICENSE.txt
tulips
roses
```

O código abaixo cria o `Dataset` de nome `list_ds`, que contém os caminhos e nomes de todos os arquivos de imagens do conjunto de dados.

Após criar o `Dataset` são apresentados os primeiros 10 arquivos da lista.

```
# Obtém lista de arquivos no diretório "flowers_root"
list_ds = tf.data.Dataset.list_files(str(flowers_root/'*/*'))

# Apresenta os primeiros 10 arquivos da lista
for f in list_ds.take(10):
    print(f.numpy())
```

```
b'/root/.keras/datasets/flower_photos/sunflowers/
184683023_737fec5b18.jpg'
b'/root/.keras/datasets/flower_photos/roses/2392457180_f02dab5c65.jpg'
b'/root/.keras/datasets/flower_photos/daisy/18622672908_eab6dc9140_n.j
pg'
b'/root/.keras/datasets/flower_photos/daisy/9158041313_7a6a102f7a_n.jp
g'
b'/root/.keras/datasets/flower_photos/dandelion/138166590_47c6cb9dd0.j
pg'
b'/root/.keras/datasets/flower_photos/roses/5402157745_a384f0583d_n.jp
g'
b'/root/.keras/datasets/flower_photos/dandelion/5600240736_4a90c10579_
n.jpg'
b'/root/.keras/datasets/flower_photos/tulips/8713391394_4b679ea1e3_n.j
pg'
b'/root/.keras/datasets/flower_photos/roses/3109712111_75cea2dee6.jpg'
b'/root/.keras/datasets/flower_photos/roses/16018886851_c32746cb72.jpg
'
```

- O método `list_files()` serve para criar um Dataset a partir de uma lista de arquivos de um diretório cujos nomes seguem um padrão fornecido como argumento para o método. Nesse exemplo, se usa como argumento `'*/*'`, o que representa selecionar todos os arquivos de todos os subdiretórios.

Tendo o Dataset `lista_ds`, que contém a lista dos caminhos e nomes dos arquivos das imagens pode-se criar o Dataset com os dados.

No código abaixo é definida a função `process_path()`. Essa função realiza as seguintes tarefas:

- recebe um caminho indicando um arquivo de imagem e carrega esse arquivo;
- extrai a classe à qual pertence a flor, usando para isso a string com o caminho do arquivo.

A função `process_path()` ao ser chamada com a lista de arquivos retorna o dataset com os pares `(image, label)`.

Para ler os arquivos usa-se a função `tf.io.read_file(nome)` do TensorFlow, onde `nome` é o caminho com o nome do arquivo a ser lido.

Para ler as classes usa-se a função `tf.strings.split(string, sep)`, que separa a string fornecida, usando como indicador para a separação o símbolo fornecido no argumento `sep`. No exemplo é selecionada a segunda string a partir do final após o símbolo "\".

```
# Função process_path que le arquivo de imagem e obtém a sua classe
def process_path(file_path):
    label = tf.strings.split(file_path, sep='/')[-2]
    return tf.io.read_file(file_path), label

# Cria dataset com as imagens e as suas classes
labeled_ds = list_ds.map(process_path)
labeled_ds
```

```
<_MapDataset element_spec=(TensorSpec(shape=(), dtype=tf.string,
name=None), TensorSpec(shape=(), dtype=tf.string, name=None))>
```

Para verificar se tudo foi realizado de forma correta, o código a seguir obtém uma imagem do Dataset juntamente com a sua classe usando o método `take()`.

A imagem é retornada como uma lista de strings (codificação "base64"). Se quisermos visualizar a imagem devemos decodificá-la usando o método `tf.io.decode_image()`, conforme mostrado abaixo.

```
# Seleciona primeiro exemplo e par imagem-classe do dataset
for image_raw, label_text in labeled_ds.take(1):
    print(repr(image_raw.numpy()[:100]))
    print()
    print(label_text.numpy())

# Decodificação da imagem para poder ser visualizada
image = tf.io.decode_image(image_raw)
```

```
# Mostra imagem
plt.imshow(image)
plt.show()
```

```
b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x00\x00\x01\x00\x01\x00\
x00\xff\xdb\x00C\x00\x03\x02\x02\x03\x02\x02\x03\x03\x03\x03\x04\x03\
x03\x04\x05\x08\x05\x05\x04\x04\x05\n\x07\x07\x06\x08\x0c\n\x0c\x0c\
x0b\n\x0b\x0b\r\x0e\x12\x10\r\x0e\x11\x0e\x0b\x0b\x10\x16\x10\x11\x13\
x14\x15\x15\x15\x0c\x0f\x17\x18\x16\x14\x18\x12\x14\x15\x14\xff\xdb\
x00C\x01\x03\x04\x04\x05\x04\x05'
```

```
b'tulips'
```

5. Criar lotes de exemplos para treinamento

Para dividir os exemplos em lotes para serem usados no treinamento da RNA, o módulo `tf.data` fornece meios de criar lotes de exemplos de um objeto `Dataset`.

5.1 Lotes simples

Para criar um lote de n exemplos de dados de um objeto `Dataset` usa-se o método `batch()`. Com esse método todos os exemplos devem ter a mesma dimensão.

O código abaixo apresenta um exemplo de definir lotes de um objeto `Dataset`. Nesse exemplo, primeiramente é criado um conjunto de dados com duas sequência de números e depois são defindos os lotes.

```
# Cria série de números crescentes de 0 a 100
inc_dataset = tf.data.Dataset.range(100)

# Cria série de números decrescentes de 0 a -100
dec_dataset = tf.data.Dataset.range(0, -100, -1)

# Cria dataset com as duas séries
dataset = tf.data.Dataset.zip((inc_dataset, dec_dataset))

# Cria novo dataset com lotes de 3 exemplos
batched_dataset = dataset.batch(3)

# Mostra 5 lotes
for batch in batched_dataset.take(5):
    print([arr.numpy() for arr in batch])
```

```
[array([0, 1, 2]), array([ 0, -1, -2])]
[array([3, 4, 5]), array([-3, -4, -5])]
[array([6, 7, 8]), array([-6, -7, -8])]
[array([ 9, 10, 11]), array([ -9, -10, -11])]
[array([12, 13, 14]), array([-12, -13, -14])]
```

- Observa-se que cada exemplo de treinamento é composto por dois números, sendo um da série crescente e outro da série descrescente.

Para vermos como o Dataset ficou após a divisão em lotes podemos fazer:

```
batched_dataset.element_spec
```

```
(TensorSpec(shape=(None,), dtype=tf.int64, name=None),
 TensorSpec(shape=(None,), dtype=tf.int64, name=None))
```

- Observe que o padrão do método `batch()` é retornar lotes de tamanho `None` (desconhecido), porque o último lote pode não estar completo. Como, ocorre no exemplo acima, pois com lotes de 3 elementos em um conjunto de 100 elementos resulta em 33 lotes de 3 elementos e um lote de 1 único elemento.

Se for desejado, isso pode ser corrigido usando o argumento `drop_remainder` para ignorar o último lote e, assim, obter todos os lotes de mesmo tamanho. Abaixo segue um exemplo.

```
# Cria novo dataset com lotes de 3 exemplos descartando o último lote
batched_dataset = dataset.batch(3, drop_remainder=True)
batched_dataset.element_spec
```

```
(TensorSpec(shape=(3,), dtype=tf.int64, name=None),
 TensorSpec(shape=(3,), dtype=tf.int64, name=None))
```

5.2 Gerando lotes para várias épocas

O módulo `tf.data` oferece duas maneiras principais de utilizar lotes de dados.

A maneira mais simples de iterar em um conjunto de dados em várias épocas é usar o método `repeat`.

Para exemplificar, vamos usar o conjunto de dados Titanic lendo as suas linhas e não carregando-o como um arquivo CSV.

```
# Define nome e URL do arquivo de dados
titanic_file = tf.keras.utils.get_file("train.csv",
"https://storage.googleapis.com/tf-datasets/titanic/train.csv")

# Cria Dataset a partir do arquivo de dados
titanic_lines = tf.data.TextLineDataset(titanic_file)

print(len(list(titanic_lines)))
```

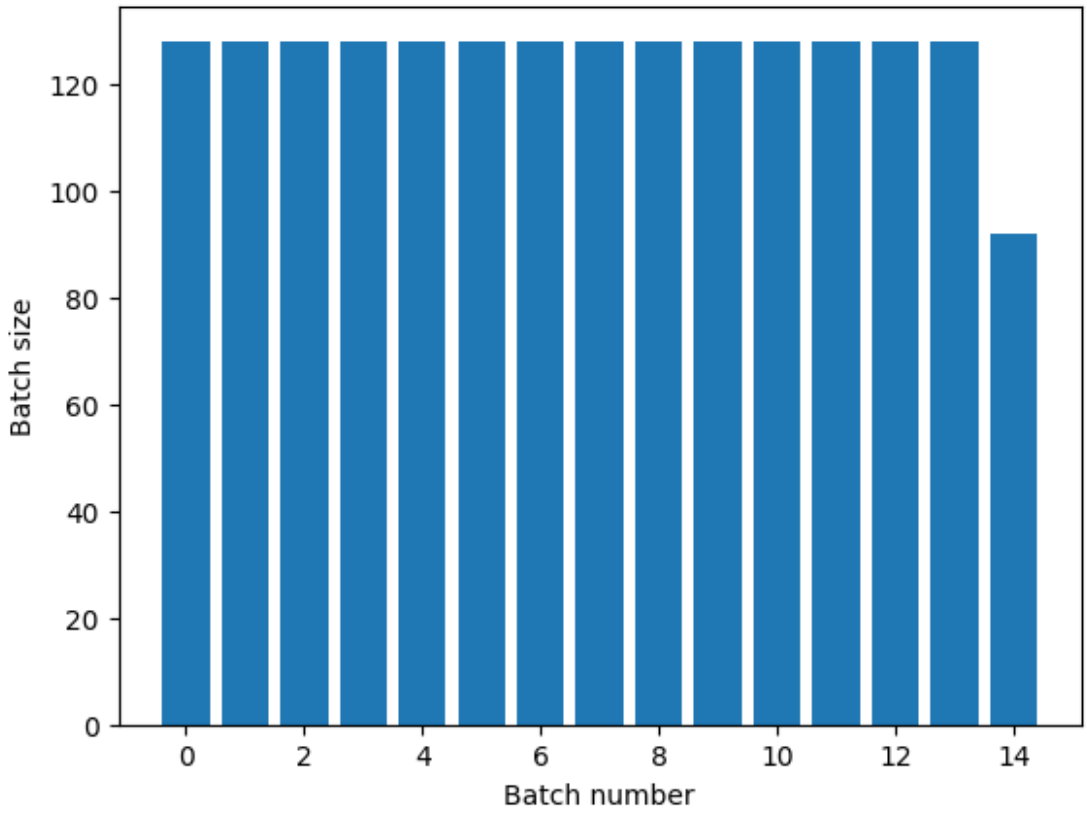
```
628
```

A função `plot_batch_sizes()`, definida abaixo, mostra um gráfico do número de elementos dos lotes gerados de um Dataset.

```
def plot_batch_sizes(ds):
    batch_sizes = [batch.shape[0] for batch in ds]
    plt.bar(range(len(batch_sizes)), batch_sizes)
    plt.xlabel('Batch number')
    plt.ylabel('Batch size')
```

Aplicar o método `repeat` antes de definir o tamanho do lote repete os dados indefinidamente. Assim, um comando `batch` aplicado após o comando `repeat` irá produzir lotes que ultrapassam os limites da época, como no exemplo a seguir.

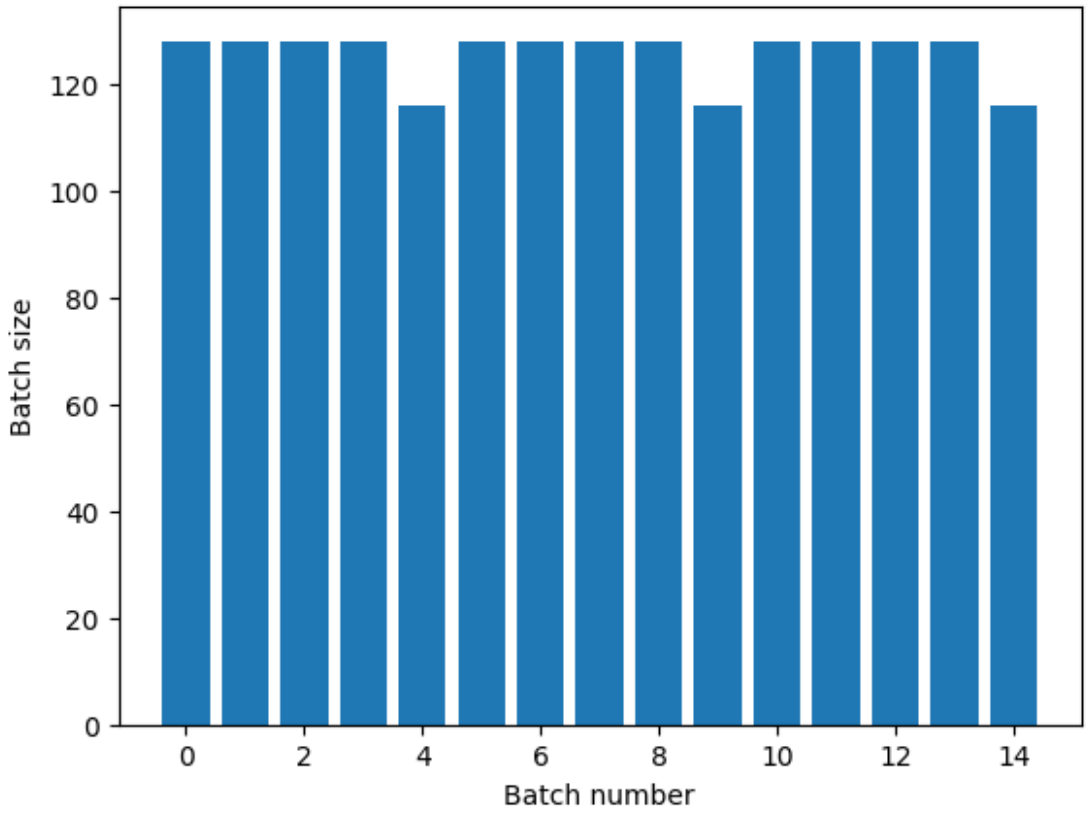
```
# Gera lotes
titanic_batches = titanic_lines.repeat(3).batch(128)

# Mostra tamanho dos lotes gerados
plot_batch_sizes(titanic_batches)
```



Esse conjunto de dados Titanic possui 630 exemplos (linhas de dados). Assim, com lotes de 128 exemplos cada, tem-se no máximo 4,9 lotes possíveis com dados diferentes e no caso foram gerados 14 lotes.

Se for necessário uma separação de época clara, deve-se colocar o método `batch` antes do método `repeat`, como mostrado a seguir.

```
# Gera lote de dados
titanic_batches = titanic_lines.batch(128).repeat(3)

# Mostra tamanho dos lotes gerados
plot_batch_sizes(titanic_batches)
```



Nesse caso, observa-se que são gerados 4,9 lotes sem repetição dos dados 3 vezes.

5.3 Embaralhamento aleatório dos dados

O método `shuffle(n)` mantém "n" exemplos em um "buffer" e seleciona os exemplos de forma aleatória entre os elementos desse "buffer".

Observa-se que quanto maior o tamanho do "buffer" melhor a seleção aleatória, mas pode ocupar muita memória e um grande tempo para ser preenchido.

O código abaixo carrega os dados na forma de linhas de texto do conjunto de dados "Titanic".

Para facilitar a visualização de como funciona esse método, é adicionado um índice aos exemplos do `Dataset`..

```
# Carrega arquivo de dados
titanic_file = tf.keras.utils.get_file("train.csv",
"https://storage.googleapis.com/tf-datasets/titanic/train.csv")

# Cria dataset com as linhas na forma de texto
lines_ds = tf.data.TextLineDataset(titanic_file)

# Cria dataset com índices
#indice_ds = tf.data.experimental.Counter()
indice_ds = tf.data.Dataset.counter()
```

```
# Cria dataset com índices e as linhas de texto
dataset = tf.data.Dataset.zip((indice_ds, lines_ds))

# Embaralha dados e cria lote de 10 exemplos
dataset = dataset.shuffle(buffer_size=100)
dataset = dataset.batch(10)
dataset.element_spec
```

```
(TensorSpec(shape=(None,), dtype=tf.int64, name=None),
 TensorSpec(shape=(None,), dtype=tf.string, name=None))
```

Na medida em que `buffer_size` é de 100 linhas, e o tamanho do lote é 10, o primeiro lote contém elementos de índices no máximo até 110.

O código abaixo mostra os exemplos (linhas do Datset) selecionados no lote.

```
# Instancia o Dataset com um gerador e executa o mesmo
n, line_batch = next(iter(dataset))

# Mostra resultados
print('Índices das linhas do conjunto de dados selecionadas no lote:',
n.numpy())
print('\nLinhas:\n', line_batch)
```

```
Índices das linhas do conjunto de dados selecionadas no lote: [  7  58
99  91  44  35  53 100  45   1]

Linhas:
 tf.Tensor(
[b'1,female,27.0,0,2,11.1333,Third,unknown,Southampton,n'
 b'1,male,32.0,0,0,56.4958,Third,unknown,Southampton,y'
 b'1,male,24.0,0,0,7.1417,Third,unknown,Southampton,y'
 b'0,male,21.0,0,0,7.925,Third,unknown,Southampton,y'
 b'1,male,28.0,0,0,35.5,First,C,Southampton,y'
 b'1,female,3.0,1,2,41.5792,Second,unknown,Cherbourg,n'
 b'0,male,19.0,0,0,8.1583,Third,unknown,Southampton,y'
 b'0,male,45.0,0,0,6.975,Third,unknown,Southampton,y'
 b'1,female,21.0,0,0,10.5,Second,unknown,Southampton,y'
 b'0,male,22.0,1,0,7.25,Third,unknown,Southampton,n'], shape=(10,),
dtype=string)
```

6. Transformação dos dados

Existem diversos métodos para transformar (processar) os dados de um objeto `Dataset`. Os métodos a serem utilizados dependem do tipo de dados.

Para dados estruturados, do tipo encontrado em arquivos tipo CSV, já vimos como utilizar o Pandas e também existe o módulo `tf.feature_column` do TensorFlow.

O método `map(f)` pode ser usado para qualquer tipo de dado. Esse método transforma os dados de um Dataset aplicando a função `f` em cada elemento do Dataset.

A função `f`, aplica pelo método `map(f)`, deve receber um tensor que representa um único elemento do Dataset e retorna outro tensor que representa também um único elemento. Essa função deve usar operações do TensorFlow para realizar as transformações desejadas.

6.1 Decodificação e redimensionamento de imagens

Quando se usa imagens reais para treinar uma RNA, geralmente é necessário redimensionar as imagens e normalizar os seus pixels.

Como exemplo de transformação de imagens vamos usar o conjunto de dados "flowers" usando o método `map`.

O código abaixo constrói esse conjunto de dados usando a lista dos arquivos no Dataset `list_ds`, como já visto.

```
list_ds = tf.data.Dataset.list_files(str(flowers_root/'*/*'))
```

- Observe que o caminho `flowres_root` já foi definido no item 4.6.

Para transformar os dados, a primeira etapa é criar uma função que manipula os elementos do Dataset.

Para exemplificar, é criada a função `transform_image`, que recebe o caminho com o nome do arquivo da imagem e realiza as seguintes operações:

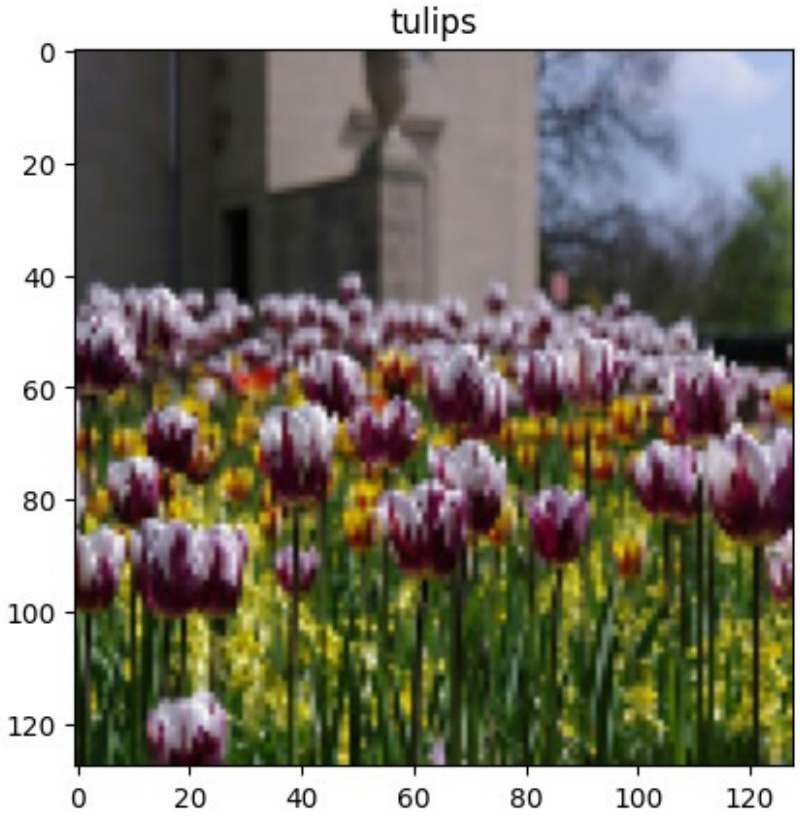
- Cria as classes das flores a partir do nome dos subdiretórios onde estão os arquivo com as imagens;
- Carrega o arquivo da imagem em memória;
- Decodifica os dados recriando a imagem original;
- Converte a imagem para números do tipo float32;
- Redimension a imagem;
- Retorna a imagem e a classe.

```
# Define função transform_image
def transform_image(filename):
    # Cria a classe dos dados
    label = tf.strings.split(filename, '/')[-2]

    # Carrega dados do arquivo em formato base 64
    image = tf.io.read_file(filename)

    # Decodifica imagem para formato jpg
    image = tf.image.decode_jpeg(image)

    # Converte imagem para tipo float32
    image = tf.image.convert_image_dtype(image, tf.float32)

    # Redimensiona imagem
```

```
    image = tf.image.resize(image, [128, 128])

    # Normaliza os pixles
    image = image/255.

    return image, label
```

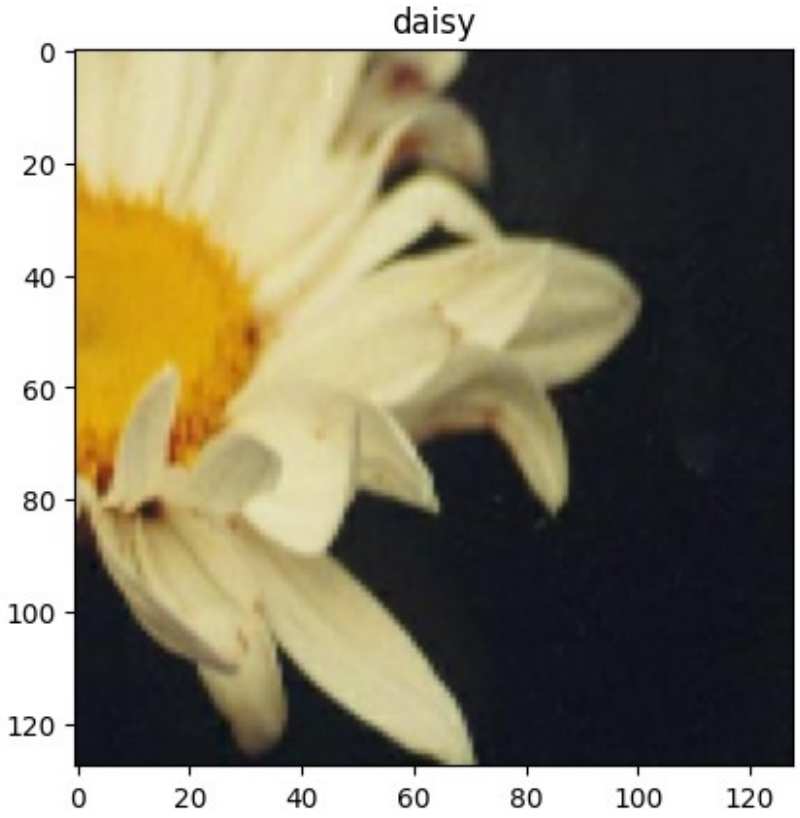
A função `transform_imagem` é testada na célula abaixo.

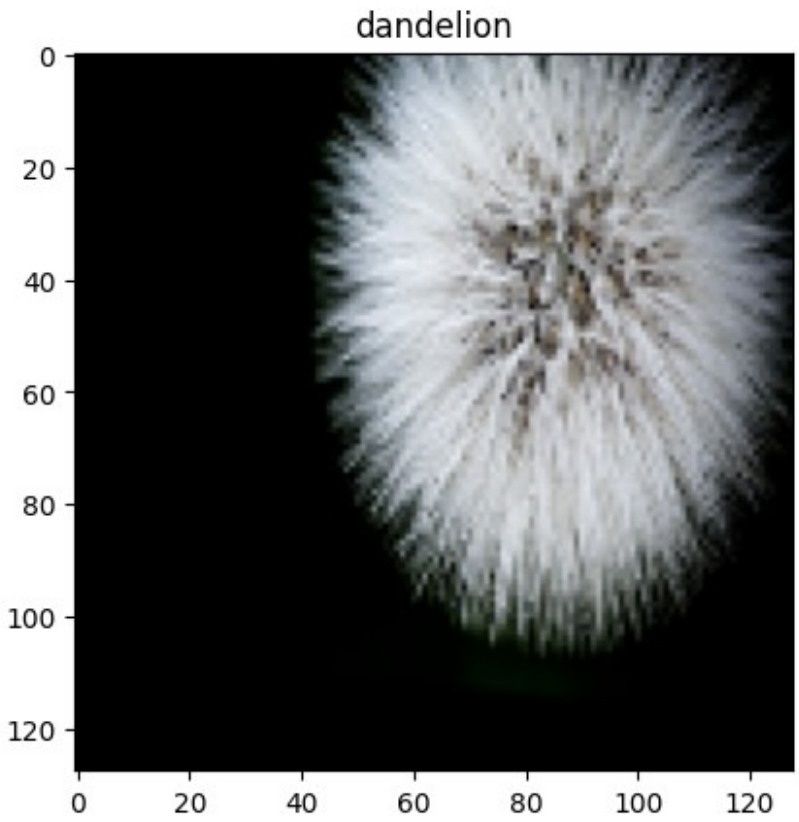
```
# Define caminho e nome da imagem pegando primeiro elemento do Dataset list_ds
file_path = next(iter(list_ds))

# Processa imagem com a função transform_imagem
image, label = transform_image(file_path)

# Define função para mostrar imagem
def show(image, label):
    plt.figure()
    plt.imshow(image*255)
    plt.title(label.numpy().decode('utf-8'))

# Mostra imagem usando a função show
show(image, label)
```



Usando a função `transform_image`, aplicada com o método `map()`, cria-se um novo Dataset com as imagens transformadas prontas para serem usadas pela RNA. Essa operação é realizada no código abaixo.

```
# Transforma imagens com função parse_images
images_ds = list_ds.map(transform_image)

# Apresenta os dois primeiros exemplos
for imagei, labeli in images_ds.take(2):
    show(imagei, labeli)
```





6.2 Janelamento de séries temporais

Dados de séries temporais devem manter a sequência no tempo inalterada, assim, eles não devem ser embaralhados antes de gerar os lotes de exemplos de treinamento.

Um exemplo de treinamento de uma série temporal para ser usado por uma rede neural, consiste de algumas amostras da sequência adquiridas em um determinado intervalo de tempo.

Observa-se que uma série temporal pode ser analisada com uma rede neural com camadas recorrentes ou com uma rede neural com camadas convolucionais de 1D.

Vamos usar uma série temporal simples como exemplo de criar um `Dataset`.

```
# Cria uma série temporal simples
serie_ds = tf.data.Dataset.range(100000)
```

Janelas de dados com uso do método `batch`

A abordagem mais simples para criar exemplos de treinamento de uma série temporal é agrupar os dados em lotes para isso, usa-se o método `batch` escolhendo o número de amostras da série em cada exemplo de treinamento.

```
# Cria Datset com lotes de 10 elementos
batches_ds = serie_ds.batch(10, drop_remainder=True)
```

```
# Mostra 5 primerios lotes
for batch in batches_ds.take(5):
    print(batch.numpy())
```

Ressalta-se que essa forma de organizar dados de séries temporais não é muito útil porque em geral deseja-se treinar uma RNA para fazer previsões. Assim, os dados de treinamento devem possuir entradas e saídas.

Janela de dados para realizar previsão de alguns passos no futuro

Em geral deseja-se realizar previsões da série para alguns instantes de tempo no futuro, assim, a saída desejada deve conter amostras da série alguns instantes de tempo no futuro em relação às amostras contidas na entrada correspondente.

Para treinar uma RNA para realizar previsões k instantes de tempo no futuro, baseado em uma série temporal, um exemplo de treinamento deve ser composto por:

- **Entrada**: conjunto de amostras da série no intervalo de tempo de $(n-m)T$ até nT, onde m é o comprimento da janela no passado, T é o intervalo de tempo entre cada amostra e nT é o tempo "presente".
- **Saída desejada**: elementos da série nos instantes de tempo $(n+1)T$ até $(n+k)T$, onde k é o números de instantes no futuro para os quais deseja-se realizar previsões.

Na célula abaixo é apresentado como criar esse tipo de exemplo de treinamento. Define-se uma função que recebe como entrada lotes do Dataset `series_ds`, criados definindo o tamanho da janela desejada, m, e quantos passos no futuro deseja-se prever.

```
# Define comprimento da janela e quantas previsões deseja-se fazer no
futuro
feature_length = 8 # m da fórmula acima
label_length = 2 # k da fórmula acima

# Cria Dataset com lotes de tamanho "m" (entradas)
features_ds = serie_ds.batch(feature_length, drop_remainder=True)

# Cria Dataset com o número de saídas futuras desejadas ("k")
labels_ds = serie_ds.batch(feature_length).skip(1).map(lambda labels:
labels[:-(feature_length-label_length)])

# Cria Daset final unindo os dois Dataset criados com as entradas e as
saídas desejadas
predict_k_steps_ds = tf.data.Dataset.zip((features_ds, labels_ds))

# Mostra 3 exemplos de treinamento
for features, label in predict_k_steps_ds.take(3):
    print(features.numpy(), " => ", label.numpy())
```

- O método `skip` aqui define o número de `feature_lengths` em que a janela é deslocada para cada exemplo.

Janela de dados com uso do método `window`

A forma anterior usando o método `batch` funciona, mas o método `window` é feito especialmente para gerar janelas de dados.

O método `window` permite um controle melhor de como criar as janelas de dados. Porém, tem que se tomar cuidado porque ele retorna um `Datset` de `Datasets`.

```
# Define tamanho da janela de dados
window_size = 5

# Define passo de deslocamento da janela
d = 1

# Cria objeto para gerar as janelas
windows_ds = serie_ds.window(window_size, shift=d)
for sub_ds in windows_ds.take(6):
    print(sub_ds)
    window_data = [element.numpy() for element in sub_ds]
    print(window_data)
```

Observe que foram criados 6 Datasets de dados e não é isso que queremos.

O método `flat_map` pode ser usado para transformar um Dataset de Datasets em um único Dataset. Porém ao fazer isso todos os elementos do Dataset são unidos em um único elemento.

Para separar cada exemplo de treinamento pode-se usar o método `batch` nesse novo conjunto de dados. Essas operações podem ser realizadas de acordo com o código da célula a seguir.

```
# Define função para separar os exemplos de treinamento gerados
def sub_to_batch(sub):
    return sub.batch(window_size, drop_remainder=True)

# Une todos os elementos do Datset e gera exemplos de treinamento com
o método window e a função sub_to_batch
for example in windows_ds.flat_map(sub_to_batch).take(10):
    print(example.numpy())
```

Além do argumento `shift`, o método `window` possui o argumento `stride`. Para entender o que fazem esses argumentos vamos criar exemplos de treinamento com alguns valores de `shift` e `stride`.

A função `make_window_dataset` agrupa todas as operações para criar lotes de janelas de uma série temporal.

```
# Função para criar lotes de janelas de uma série temporal
def make_window_dataset(ds, window_size=5, shift=1, stride=1):
    windows = ds.window(window_size, shift=shift, stride=stride)

    def sub_to_batch(sub):
```

```
        return sub.batch(window_size, drop_remainder=True)

    windows = windows.flat_map(sub_to_batch)
    return windows

# Instancia objeto Dataset com função make_window_dataset
ds = make_window_dataset(serie_ds, window_size=5, shift=3, stride=1)

# Gera exemplos de treinamento
for example in ds.take(10):
    print(example.numpy())
```

Para finalizar, falta criar as saídas dos exemplos de treinamento. Para isso podemos criar uma função que separa as saídas das entradas dos exemplos de treinamento criados com a função `make_window_dataset`.

```
# Define tamanho da janela (m) e numero de previsões no futuro (k)
feature_length = 5 # m da fórmula acima
label_length = 1 # k da fórmula acima

# Deve-se inicialmente criar janelas com a quantidade de amostras
iguais ao m+k
total_length = feature_length + label_length

# Cria exemplos de treinamento que contém as entradas e as saídas
ds = make_window_dataset(serie_ds, window_size=total_length, shift=1,
stride=1)

# Define função para separar entradas das saídas
def labels(ds):
    saida = ds[-label_length:]
    entrada = ds[:feature_length]
    return entrada, saida

# Executa função para separar entradas das saídas
dense_labels_ds = ds.map(labels)

# Mostra 3 exemplos de treinamento
for inputs, labels in dense_labels_ds.take(3):
    print(inputs.numpy(), "=>", labels.numpy())
```

6.3 Conjunto de dados com classes desbalanceadas

Vimos anteriormente um método para treinar uma RNA com um conjunto de dados desbalanceado usando pesos para as classes.

Existem outras formar de lidar com esse tipo de dados sem a necessidade de usar pesos para as classes. Essas formas são baseadas em criar lotes de dados com as classes balanceadas → o módulo `tf.data` fornece métodos para fazer isso.

Para exemplicar essas formas vamos utilizar o conjunto de dados "The credit card fraud". Esse conjunto de dados pode ser carregado diretamente do TensorFlow. Uma descrição desses dados pode ser vista em https://www.kaggle.com/mlg-ulb/creditcardfraud.

Vamos inicialmente carregar esse conjunto de dados em um DataFrame Pandas para poder visualizá-lo e entender melhor os dados.

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

print('Dimensão do cojunto de dados:', df.shape)
```

Esse conjunto de dados possui 284.807 exemplos, sendo que cada exemplo possui 30 características e uma saída que representa as duas classes: dados sem fraude (classe = 0) e dados com fraude (classe = 1).

Vamos calcular as estatísticas básicas de cada coluna (característica) do conjunto de dados.

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

Carregar conjunto de dados para permitir criar lotes com classes balanceadas

Vamos carregar esse conjunto de dados de forma a carregar os dados lote por lote.

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

```
# Define local do conjunto de dados
zip_path = tf.keras.utils.get_file(

origin='https://storage.googleapis.com/download.tensorflow.org/data/
creditcard.zip',
    fname='creditcard.zip',
    extract=True)

# Carrega conjunto de dados e descompacta arquivo
csv_path = zip_path.replace('.zip', '.csv')
```

Criar objeto Dataset a partir do conjunto de dados

Esse Dataset é criado e no mesmo comando são realizadas as seguintes definições:

- Definição do tamanho dos lotes;
- Indicada a coluna onde se encontram as saídas desejadas;
- Transforma os tipos de dados em reais (entradas) e inteiro (saída).

```
creditcard_ds = tf.data.experimental.make_csv_dataset(
    csv_path, batch_size=1024, label_name="Class",
```

```
    shuffle = False, # Embaralhamento de dados não funciona com esse
tipo de processo
    column_defaults=[float()]*30+[int()]) # Define os tipos de dados
das colunas (30 reais e 1 inteira)
```

Para verificar o conteúdo de um lote de dados, tem-se:

```
# Verifica tipo dos elementos
creditcard_ds.element_spec

print('Estrutura de um lote de dados:\n', creditcard_ds)
print('\nExemplo de um lote:')
for lote in creditcard_ds.take(1):
    print(format(lote))

print('\n')
for features, labels in creditcard_ds.take(1):
    for key, value in features.items():
        features[key] = value.numpy()
    labels = labels.numpy()
    print(features, labels)
```

Cálculo do número de exemplos de cada classe

A função `count()` definida abaixo conta o número de exemplos de cada classe.

```
def count(counts, batch):
    features, labels = batch
    class_1 = labels == 1
    class_1 = tf.cast(class_1, tf.int32)

    class_0 = labels == 0
    class_0 = tf.cast(class_0, tf.int32)

    counts['class_0'] += tf.reduce_sum(class_0)
    counts['class_1'] += tf.reduce_sum(class_1)

    return counts
```

Executa função `count()` para calcular o número de exemplos de cada classe. Para isso é utilizado o método `reduce` que aplica a função `count()` no Dataset.

```
counts = creditcard_ds.take(10).reduce(initial_state={'class_0': 0,
'class_1': 0}, reduce_func = count)

counts = np.array([counts['class_0'].numpy(),
counts['class_1'].numpy()]).astype(np.float32)

fractions = counts/counts.sum()
```

```
print('Fração de dados sem fraude:', fractions[0])
print('Fração de dados com fraude:', fractions[1])
```

- O método `reduce` chama a função `count()` sucessivamente para cada elemento do conjunto de dados de entrada até que o conjunto de dados se esgote, agregando informações em seu estado interno. O argumento `initial_state` é usado para definir o estado inicial da variável desejada e o estado final é retornado como resultado
- Observe como as duas classes de dados são altamente desbalanceadas.

Balanceamento das classes nos lotes de exemplos de treinamento

O módulo `tf.data` fornece duas abordagens para criar lotes com exemplos das classes balanceadas:

- Re-amostragem do Dataset
- Rejeição de amostras

Abordagem #1: Amostragem do Dataset

Nessa forma, cada lote de dados criado é amostrado do Dataset usando o método `sample_from_datasets`.

Essa forma somente pode ser utilzada quando se tem dois Datasets separados, sendo um para cada classe. Assim, deve-se primeiramente criar os dois Datastes a partir do Dataset original. Para isso usa-se o método `filter`, da seguinte forma:

```
# Cria Dataset com os dados sem fraude (negativo, classe = 0)
negativos_ds = creditcard_ds.unbatch().filter(lambda features, label:
label==0).repeat()

# Cria Dataset com os dados com fraude (positivo, classe = 1)
positivos_ds = creditcard_ds.unbatch().filter(lambda features, label:
label==1).repeat()
```

- O método `unbatch` une todos os elementos de um Dataset em um único elemento.
- O método `filter` aplica uma função especificada no Dataset. Essa função deve representar uma condição e os elementos que satisfazem essa condição são slecionados.

Vamos verificar o resultado dessas operações visualizando as classes dos exemplos em cada um desses dois Datasets criados.

```
# Mostra um lote com 10 exemplos sem fraude
for features, label in negativos_ds.batch(10).take(1):
    print('Classes dos exemplos do Dataset sem fraude:',
label.numpy())
```

```
# Mostra um lote com 10 exemplos com fraude
for features, label in positivos_ds.batch(10).take(1):
    print('Classes dos exemplos do Dataset com fraude:',
label.numpy())
```

Para criar lotes com exemplos das duas classes usa-se o método `sample_from_datasets`, passando a fração do número de exemplos desejados de cada clase, da seguinte forma:

```
# Dataset que cria lotes de elementos com as classes desbalanceadas
balanced_ds = tf.data.Dataset.sample_from_datasets(
              [negativos_ds, positivos_ds], [0.5,
0.5]).shuffle(100).batch(10)
```

- Cada lote contém 10 elementos.
- Nesse caso foi escolhido ter metade dos exemplos do lote de cada classe.

Para verificar o resultado, vamos mostrar as classes de 8 lotes com exemplos cada.

```
# Mostra 8 primeiros lotes do Dataset balanced
for features, labels in balanced_ds.take(8):
    print(labels.numpy())
```

Abordagem #2: Rejeição de amostras

Um problema com a abordagem anterior, usando os métodos `sample_from_datasets` e `filter`, é que precisa de um Dataset separado para cada classe, resultando na necessidade de replicar os dados e, assim, ocupa mais memória.

O método `rejection_resample` pode ser aplicado em um Dataset para balancear as classes ao mesmo tempo em que carrega os lotes de dados durante o treinamento. Nessa abordagem elementos são eliminados do lote até obter o número de classes balanceado.

Nessa abordagem, o método `experimental.rejection_resample` recebe a função `class_func` como argumento. Essa função `class_func` é aplicada em cada elemento do lote para determinar de qual classe aquele elemento pertence para poder realizar o balanceamento.

Observa-se que essa abordagem somente funciona se os lotes forem os mesmos em cada época. Assim, nessa abordagem não se pode usar o método `shuffle` para embaralhar os exemplos.

O Dataset `creditcard` já está preparado para retornar lotes com pares de entradas e saídas `(features, label)`, assim, a função `class_func` deve ser definida para retornar somente as saídas. Essa função é bem simples e está definida na célula a seguir.

```
# Função que retorna classe dos exemplos de treinamento
def class_func(features, label):
    return label
```

O amostrador de elementos do lote é um objeto da classe `rejection_resample`. Esse objeto precisa de um objetivo para a distribuição de cada classe nos lotes e a forma como essa distribuição é descrita.

Um exemplo de como instanciar um objeto `rejection_resample` é definida na célula abaixo.

```
# Instancia objeto rejection_resample para balancear lote
resampler = creditcard_ds.unbatch().rejection_resample(
            class_func, target_dist=[0.5, 0.5],
initial_dist=fractions)
```

- No caso tem-se cada classe com 50% dos elementos em cada lote.
- O objeto `resampler` analisa cada exemplo de treinamento individualmente, assim, para poder ser utilizado é necessário `unbatch` o Dataset antes de aplicá-lo.

Na célula a seguir é criado o Dataset para gerar os lotes com o `resampler`.

```
# Cria Dataset que gera lotes com classes balanceadas usando a
abordagem de re-amostragem
#resample_ds =
creditcard_ds.unbatch().apply(resampler).shuffle(100).batch(10)
resample_ds = resampler.shuffle(100).batch(10)

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

O `resampler` criado, retorna um par classe-exemplo (`class, example`), porém a função `class_func` já retorna as saídas (`labels`). Nesse caso cada exemplo de treinamento já possui a sua saída associada, assim, esse processo resulta em que cada elemento do lote tenha uma entrada e duas saídas. Portanto, deve-se eliminar uma das saídas e para isso pode-se usar o método `map` como mostrado a seguir.

```
# Elimina a saída repetida em cada exemplo de treinamento
balanced_ds = resample_ds.map(lambda extra_label, features_and_label:
features_and_label)

# Mostar exemplos de lotes gerados pelo Dataset
for features, labels in balanced_ds.take(3):
    print(len(features))
    print(labels.numpy())
```

7. Divisão de Datasets em dados de treinamento, validação e teste

Existem algumas formas de dividir um Dataset em Datasets de treinamento, validação e teste. Vamos ver duas dessas formas.

Outra forma de dividir os dados em vários conjuntos é realizar essa divisão antes de criar o Dataset usando, por exemplo, a função `train_test_split` da biblioteca ScikitLearn.

7.2 Forma simples

Uma forma bem simples de realizar uma divisão de dados de um Dataset é simplesmente usar o método `take` para pegar um número de exemplos (lotes) determinados de um Dataset.

Observa-se que esse método é determinístico, ou seja, sempre divide os dados da mesma forma e sempre seleciona os exemplos na mesma sequência em que estão no Dataset original.

Para exemplificar o uso dessa forma de dividir um Dataset, vamos usar o exemplo do Dataset `titanic` e dividí-lo nos conjuntos de treinamemto, validação e teste.

```
# Define nome do arquivo e URL com o arquivo de dados
titanic_file = tf.keras.utils.get_file("train.csv",
"https://storage.googleapis.com/tf-datasets/titanic/train.csv")

# Carrega dados com o Pandas
df = pd.read_csv(titanic_file, index_col=None)

# Cria dataset com o dataframe Pandas com lote de 1 exemplo
titanic_ds = tf.data.Dataset.from_tensor_slices(dict(df)).batch(1)

# Determina número de lotes nos Datasets de validação e teste
num_elements = 100

# Divide Dataset em 3
val_ds = titanic_ds.take(num_elements)
test_ds = titanic_ds.skip(num_elements).take(num_elements)
train_ds = titanic_ds.skip(2*num_elements)

# Itera nos Datasets para calcular número de lotes
test_ds_length = [i for i,_ in enumerate(test_ds)][-1] + 1
val_ds_length = [i for i,_ in enumerate(val_ds)][-1] + 1
train_ds_length = [i for i,_ in enumerate(train_ds)][-1] + 1

print('Número de lotes de treinamento:', train_ds_length)
print('Número de lotes de validação:', val_ds_length)
print('Número de lotes de teste:', test_ds_length)
```

```
Número de lotes de treinamento: 427
Número de lotes de validação: 100
Número de lotes de teste: 100
```

- Observe que no caso os lotes são de um único exemplo, então, o número de lotes é igual ao número de exemplos dos Datasets.
- Note que nesse caso temos que conhecer o número de lotes (ou exemplos) do Dataset se quisermos ter uma fração definida para os três conjuntos de dados.

7.2 Forma geral

Vamos ver uma forma mais geral de dividir um Dataset em conjuntos de treinamento e teste, que funciona para qualquer tipo de Dataset.

A primeira etapa é criar duas funções que selecionam os exemplos para serem do conjunto de treinamento ou do conjunto de teste.

Nesse exemplo, vamos agora dividir os dados nos conjuntos de treinamento e teste, sendo 80% dos dados para treinamento e 20% para teste.

Na célula abaixo são criados duas funções:

- Função `is_test()`, que seleciona 1 em cada 5 exemplos como sendo de teste;
- Função `is_train()`, que seleciona 4 em cada 5 exemplos com sendo de treinamento.

Essas funções selecionam os exemplos tendo como base um índice, que é incluído pelo método enumerate.

A função `recover` elimina o índice incluído pelo método `enumerate` e, assim, recuperr os exemplos originais.

Após criar essas funções o Dataset é dividido usando os métodos: `enumerate`, `filter` e a função `recover()`.

- O método `enumerate` introduz indices numéricos nos exemplos para permitir que sejam selecionados pelas funções `is_test()` e `is_train()`;
- O método `filter`, chama as funções `is_test()` e `is_trains`, que selecionam os exemplos com base nos seus índices;
- O método `map` chama a função `recover` para eliminar o índice incluído pelo método `enumerate`.

Obviamente que se for desejado divir os dados em três conjuntos (treinamento, validação e teste), temos que definir 3 funções para selecionar os exemplos, ou seja, temos que definir outra função, `is_val(),` do mesmo tipo que as funções `is_train()` e `is_test()`. Além disso, temos que fazer algumas alterações nessas funções.

Na célula a seguir são defindas as funções `is_test()`, `is_train()` e `recover()`.

```
# 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
```

Para exemplificar o uso dessa forma de dividir um Dataset, vamos usar novamente o Dataset `titanic` e dividí-lo nos conjuntos de treinamemto e teste.

```
# Cria Dataset de teste
test_ds = titanic_ds.enumerate().filter(is_test).map(recover)

# Cria Dataset de treinamento
train_ds = titanic_ds.enumerate().filter(is_train).map(recover)
```

Para verificar o conteúdo de um lote de dados, pode-se fazer:

```
# verifica o conteúdo de um lote do Dataset
test_ds.element_spec
```

```
{'survived': TensorSpec(shape=(None,), dtype=tf.int64, name=None),
 'sex': TensorSpec(shape=(None,), dtype=tf.string, name=None),
 'age': TensorSpec(shape=(None,), dtype=tf.float64, name=None),
 'n_siblings_spouses': TensorSpec(shape=(None,), dtype=tf.int64,
name=None),
 'parch': TensorSpec(shape=(None,), dtype=tf.int64, name=None),
 'fare': TensorSpec(shape=(None,), dtype=tf.float64, name=None),
 'class': TensorSpec(shape=(None,), dtype=tf.string, name=None),
 'deck': TensorSpec(shape=(None,), dtype=tf.string, name=None),
 'embark_town': TensorSpec(shape=(None,), dtype=tf.string, name=None),
 'alone': TensorSpec(shape=(None,), dtype=tf.string, name=None)}
```

Para verificar número de lotes do Dataset.

```
# Itera nos Datasets para calcular número de lotes
test_ds_length = [i for i,_ in enumerate(test_ds)][-1] + 1
train_ds_length = [i for i,_ in enumerate(train_ds)][-1] + 1

print('Número de lotes de treinamento:', train_ds_length)
print('Número de lotes de teste:', test_ds_length)
```

```
Número de lotes de treinamento: 501
Número de lotes de teste: 126
```

- Observe novamente que no caso os lotes são de um único exemplo, então, o número de lotes é igual ao número de exemplos dos Datasets.

8. Otimização do dataset

O uso de GPUs reduz radicalmente o tempo necessário para executar uma única etapa de treinamento. Alcançar o desempenho máximo requer um pipeline de entrada eficiente que forneça dados para a próxima etapa antes que a etapa atual seja concluída.

Existem diversas formas de acelerar a preparação de dados para o treinamento de um modelo usando um pipeline de dados.

```
import time
```

8.1 Conjunto de dados

Vamos criar um conjunto de dados e medir o desempenho do uso desses dados para várias configurações possíveis de utilização do mesmo.

Vamos criar um exemplo artificial definindo uma classe herdada de `tf.data.Dataset` chamada `ArtificialDataset`.

Este `dataset` realiza o seguinte:

- Gera amostras `num_samples`;
- Dorme algum tempo antes do primeiro item para simular a abertura de um arquivo;
- Dorme algum tempo antes de produzir cada item para simular a leitura de dados de um arquivo.

```
class ArtificialDataset(tf.data.Dataset):
    def _generator(num_samples):
        # Simulando abertura de um arquivo
        time.sleep(0.03)

        for sample_idx in range(num_samples):
            # Leitura dos dados do arquivo
            time.sleep(0.015)
            yield (sample_idx,)

    def __new__(cls, num_samples=3):
        return tf.data.Dataset.range(num_samples)
```

- Esse dataset é similar ao `tf.data.Dataset.range`, com a adição de um atraso de tempo fixo entre a leitura de cada dado.

8.2 Loop de treinamento

Vamos criar um loop de treinamento fictício que mede quanto tempo leva para iterar em um conjunto de dados.

Note que o tempo de treinamento é simulado.

```
def benchmark(dataset, num_epochs=2):
    start_time = time.perf_counter()
    for epoch_num in range(num_epochs):
        for sample in dataset:
            # Realizando uma etapa do treinamento
            time.sleep(0.01)
    print("Tempo de execução:", time.perf_counter() - start_time)
```

Modelo de referência

Vamos executar o loop de treinamento sem nenhuma otimição para ser usado cono referência.

```
benchmark(ArtificialDataset(100))
```

```
Tempo de execução: 2.0718988810003793
```

A figura abaixo (retirada de https://www.tensorflow.org/guide/images/data_performance/naive.svg) mostra como o tempo é usado nas várias etapas do processo de treinamento.

O gráfico mostra que a execução de uma etapa de treinamento envolve:

- Abrir um arquivo se ainda não tiver sido aberto
- Buscando dados no arquivo
- Usando os dados para treinamento

No entanto, em uma implementação síncrona simples como esta, enquanto o pipeline busca os dados, o modelo fica ocioso. Por outro lado, enquanto o modelo está sendo treinado, o pipeline de entrada fica ocioso. O tempo da etapa de treinamento é, portanto, a soma dos tempos de abertura, leitura e treinamento.

8.3 Pré-busca (Prefetching)

A pré-busca se sobrepõe ao pré-processamento e à execução do modelo de uma etapa de treinamento.

Enquanto o modelo executa a etapa de treinamento "i", o pipeline de entrada lê os dados da etapa `i+1` → isso reduz o tempo da etapa (em oposição à soma) do treinamento e o tempo necessário para extrair os dados.

Para realizar essa pré-busca usa-se o método `tf.data.Dataset.prefetch`.

O número de elementos na pré-busca deve ser igual (ou possivelmente maior que) ao número de lotes usados em uma época de treinamento → pode-se ajustar manualmente esse valor ou configurá-lo para `tf.data.AUTOTUNE`, ajustar o número dinamicamente em função do tempo de execução.

```
benchmark(
    ArtificialDataset(100).prefetch(tf.data.AUTOTUNE)
)
```

```
Tempo de execução: 2.0665011039982346
```

A figura abaixo mostra de tempo de execução de dados (obtida de https://www.tensorflow.org/guide/images/data_performance/prefetched.svg).

Nesse caso enquanto a etapa de treinamento está em execução para a amostra 0, o pipeline de entrada está lendo os dados da amostra 1 e assim por diante.

8.4 Paralelização da obtenção dos dados

Em um ambiente real, os dados de entrada podem ser armazenados remotamente.

Um pipeline de que funciona bem ao ler dados localmente pode apresentar gargalos na E/S ao ler dados remotamente devido às seguintes diferenças entre armazenamento local e remoto.

Além disso, depois que os bytes são carregados na memória, pode ser necessário desserializar e/ou descriptografar os dados, o que requer cálculo adicional. Essa sobrecarga está presente independentemente de os dados serem armazenados local ou remotamente, mas pode ser pior no caso remoto se os dados não forem pré-buscados de forma eficaz.

Para mitigar o impacto das várias sobrecargas de extração de dados, o método `tf.data.Dataset.interleave` pode ser usado para paralelizar a etapa de carregamento de dados.

O número de conjuntos de dados a serem sobrepostos pode ser especificado pelo argumento `cycle_length`, enquanto o nível de paralelismo pode ser especificado pelo argumento `num_parallel_calls`.

Semelhante à transformação `prefetch`, a transformação `interleave` suporta `tf.data.AUTOTUNE`, que delega a decisão sobre qual nível de paralelismo usar.

```
benchmark(
    ArtificialDataset(100).interleave(lambda _: ArtificialDataset(),
num_parallel_calls=tf.data.AUTOTUNE)
    )
```

```
Tempo de execução: 8.207120181999926
```

A figura abaixo mostra de tempo de execução de dados (obtida de https://www.tensorflow.org/guide/images/data_performance/parallel_interleave.svg).

Nesse caso enquanto a etapa de treinamento está em execução, a leitura de dois lotes é realizada em paralelo.

8.5 Paralelização da transformação dos dados

Ao preparar dados, os elementos de entrada podem precisar ser pré-processados. Para isso é usado o método `tf.data.Dataset.map`, que aplica uma função definida pelo usuário a cada elemento do conjunto de dados de entrada.

Como os elementos de entrada são independentes uns dos outros, o pré-processamento pode ser paralelizado em vários núcleos da CPU. Para tornar isso possível, semelhantemente às transformações `prefetch` e `interleave`, a transformação `map` fornece o argumento `num_parallel_calls` para especificar o nível de paralelismo.

A escolha do melhor valor para o argumento `num_parallel_calls` depende do seu hardware, das características dos seus dados de treinamento (como tamanho e formato), do custo da sua função de transfromação e de quais outros processamentos estão acontecendo na CPU ao mesmo tempo.

Uma forma simples é usar o número de núcleos de CPU disponíveis. No entanto, da mesma forma que os métodos `prefetch` e `interleave`, o `map` suporta `tf.data.AUTOTUNE` que delega a decisão sobre qual nível de paralelismo usar.

Vamos criar uma função de transformação "dummy".

```
def mapped_function(s):
    # Realiza algum cálculo de pré-processamento dos dados
    tf.py_function(lambda: time.sleep(0.03), [], ())
    return s
```

O uso dessa função de transformação sem paralelismo é realizada na célula a seguir.

```
benchmark(ArtificialDataset(100).map(mapped_function))
```

```
Tempo de execução: 8.277971031000021
```

A figura abaixo (obtida de https://www.tensorflow.org/guide/images/data_performance/sequential_map.svg) mostra o tempo de execução de cada etapa do processo.

O uso da função de pré processamento dos dados em múltiplas CPUs é realizada na célula abaixo.

```
benchmark(
    ArtificialDataset(100).map(mapped_function,
num_parallel_calls=tf.data.AUTOTUNE)
)
```

```
Tempo de execução: 5.096519664000027
```

Como mostra o gráfico (https://www.tensorflow.org/guide/images/data_performance/parallel_map.svg), as etapas de pré-processamento se sobrepõem, reduzindo o tempo total para uma única iteração.

8.6 Transformação dos dados vetorizada

Como vimos, o método `batch` cria lotes de dados em um dataset. Para acelerar o pré processamento de dados usando o método `map` com afunção de transfromação definida, deve-se primeiramente criar um lote de dados para depois aplicar a função de transfromação usando o método `map`.

Invoking a user-defined function passed into the `map` transformation has overhead related to scheduling and executing the user-defined function. Vectorize the user-defined function (that is,

have it operate over a batch of inputs at once) and apply the `batch` transformation *before* the `map` transformation.

To illustrate this good practice, your artificial dataset is not suitable. The scheduling delay is around 10 microseconds (10e-6 seconds), far less than the tens of milliseconds used in the `ArtificialDataset`, and thus its impact is hard to see.

For this example, use the base `tf.data.Dataset.range` function and simplify the training loop to its simplest form.

```
benchmark(
    ArtificialDataset(100).batch(10).map(mapped_function,
num_parallel_calls=tf.data.AUTOTUNE)
)

Tempo de execução: 0.3649481399999672
```

8.7 Uso de cache

O método `tf.data.Dataset.cache` pode armazenar em cache um conjunto de dados, na memória ou no armazenamento local.

Isso evita que algumas operações (como abertura de arquivos e leitura de dados) sejam executadas durante cada época.

```
benchmark(
    ArtificialDataset(100).map(mapped_function).cache()
    )

Tempo de execução: 4.343202855000072
```

O gráfico do tempo de execução de dados (https://www.tensorflow.org/guide/images/data_performance/cached_dataset.svg) mostra que quando se armazena em cache um conjunto de dados, as transformações antes do `cache` (como a abertura do arquivo e a leitura dos dados) são executadas apenas durante a primeira época. As próximas épocas reutilizam os dados armazenados em cache pelo método `cache`.

Se a função definida pelo usuário passada para a transformação `map` exigir muito tempo de CPU, deve-se usar `cache` após `map`, desde que o conjunto de dados resultante caiba na memória ou no armazenamento local.

Se a função de transfromação aumentar o espaço necessário para armazenar o conjunto de dados além da capacidade do cache, deve-se aplicar essa transfromação após o `cache` ou, deve-se pré-processar os dados antes do treinamento para reduzir o uso de recursos.

Usando todas as opções

Podemos usar todas as formas de acelerar o pipeline de dados para obter um desempenho melhor.

```
benchmark(

ArtificialDataset(100).prefetch(tf.data.AUTOTUNE).batch(100).map(mapped_function, num_parallel_calls=tf.data.AUTOTUNE).cache()
)

Tempo de execução: 0.06800149000002875
```

Resumo

Em resumo as práticas recomendadas para criar pipelines de entrada de dados para o TensorFlow de alto desempenho são:

- Usar a pré-busca (`prefetch´) para sobrepor o trabalho de gerar dados e de treinamento;
- Paralelizar a transformação de leitura de dados usando a transformação `interleave`;
- Paralelizar a transformação dos dados usando a função `map` definindo o argumento `num_parallel_calls`;
- Usar `cache` para armazenar dados na memória durante a primeira época;
- Vetorizar funções definidas pelo usuário passadas para a função `map`;
- Reduzir o uso de memória ao aplicar as transformações `interleave`, `prefetch` e `shuffle`.