Atividade Prática

Disciplina	**MAP Bootcamp**
Atividade	**Rede Neural – Classificação de Imagens**

1. Objetivos

O objetivo desse exercício é classificar imagens de flores.Utilizaremos um conjunto de imagens para ensinar à rede as novas classes que precisa reconhecer. Vamos usar um arquivo de fotos de flores licenciadas da creative-commons do Google. O conjunto de dados usado neste exemplo é distribuído como diretórios de imagens, com uma classe de imagem por diretório.

2. Enunciado

Essa atividade fornece um exemplo simples de como carregar um conjunto de dados de imagens usando tf.data.

```
import tensorflow as tf
```

```
AUTOTUNE = tf.data.experimental.AUTOTUNE
```

```
import IPython.display as display
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import os
```

```
tf.__version__
```

Recuperando as imagens. Antes de iniciar qualquer treinamento, você precisará de um conjunto de imagens para ensinar à rede as novas aulas que deseja reconhecer. Você pode usar um arquivo de fotos de flores licenciadas da creative-commons do Google.
O conjunto de dados usado neste exemplo é distribuído como diretórios de imagens, com uma classe de imagem por diretório.
Se quiser conferir o banco de imagens, baixe as fotos de:

http://download.tensorflow.org/example_images/flower_photos.tgz

```
import pathlib
data_dir = tf.keras.utils.get_file(origin='https://storage.googleapis.com/download.tensorfl
                                   fname='flower_photos', untar=True)
data_dir = pathlib.Path(data_dir)
```

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

Após o download (218MB), você deve ter uma cópia das fotos da flor disponível.
O diretório contém 5 subdiretórios, um por classe:

```
image_count = len(list(data_dir.glob('*/*.jpg')))
image_count
```

O count retornará 3670 imagens.

```
CLASS_NAMES = np.array([item.name for item in data_dir.glob('*') if item.name != "LICENSE.txt"])
CLASS_NAMES
```

O retorno será:
array(['sunflowers', 'daisy', 'roses', 'tulips', 'dandelion'], dtype='<U10')

Cada diretório contém imagens desse tipo de flor. Aqui estão algumas rosas:

```
roses = list(data_dir.glob('roses/*'))

for image_path in roses[:3]:
    display.display(Image.open(str(image_path)))
```

Carregar usando keras.preprocessing

Uma maneira simples de carregar imagens é usar tf.keras.preprocessing.

```
# The 1./255 is to convert from uint8 to float32 in range [0,1].
image_generator = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255)
```

Defina alguns parâmetros para a carga:

```
BATCH_SIZE = 32
IMG_HEIGHT = 224
IMG_WIDTH = 224
STEPS_PER_EPOCH = np.ceil(image_count/BATCH_SIZE)
```

```
train_data_gen = image_generator.flow_from_directory(directory=str(data_dir),
                                                     batch_size=BATCH_SIZE,
                                                     shuffle=True,
                                                     target_size=(IMG_HEIGHT, IMG_WIDTH),
                                                     classes = list(CLASS_NAMES))
```

O resultado será 3670 imagens pertencendo a 5 classes.

Inspecione o lote:

```
def show_batch(image_batch, label_batch):
  plt.figure(figsize=(10,10))
  for n in range(25):
      ax = plt.subplot(5,5,n+1)
      plt.imshow(image_batch[n])
      plt.title(CLASS_NAMES[label_batch[n]==1][0].title())
      plt.axis('off')
```

```
image_batch, label_batch = next(train_data_gen)
show_batch(image_batch, label_batch)
```

Carregar usando tf.data

O método keras.preprocessing acima é conveniente, mas tem três desvantagens:

1) É lento. Veja a seção de desempenho abaixo.
2) Falta controle refinado.
3) Não está bem integrado ao restante do TensorFlow.

Para carregar os arquivos como um tf.data.Dataset, primeiro crie um conjunto de dados dos caminhos do arquivo:

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

```
for f in list_ds.take(5):
  print(f.numpy())
```

Escreva uma função pure-tensorflow que converte um caminho de arquivo em um par (img, label):

```
def get_label(file_path):
  # convert the path to a list of path components
  parts = tf.strings.split(file_path, os.path.sep)
  # The second to last is the class-directory
  return parts[-2] == CLASS_NAMES
```

```
def decode_img(img):
  # convert the compressed string to a 3D uint8 tensor
  img = tf.image.decode_jpeg(img, channels=3)
  # Use `convert_image_dtype` to convert to floats in the [0,1] range.
  img = tf.image.convert_image_dtype(img, tf.float32)
  # resize the image to the desired size.
  return tf.image.resize(img, [IMG_HEIGHT, IMG_WIDTH])
```

```
def process_path(file_path):
  label = get_label(file_path)
  # load the raw data from the file as a string
  img = tf.io.read_file(file_path)
  img = decode_img(img)
  return img, label
```

Use Dataset.map para criar um conjunto de dados de imagem, pares de rótulos:

```
# Set `num_parallel_calls` so multiple images are loaded/processed in parallel.
labeled_ds = list_ds.map(process_path, num_parallel_calls=AUTOTUNE)
```

```
for image, label in labeled_ds.take(1):
  print("Image shape: ", image.numpy().shape)
  print("Label: ", label.numpy())
```

O resultado será:

Image shape: (224, 224, 3)

Label: [False False True False False]

Métodos básicos para treinamento

Para treinar um modelo com esse conjunto de dados, você deseja os dados:

- Estar bem embaralhado.
- Para ser agrupado.

- Os lotes devem estar disponíveis o mais rápido possível.

Esses recursos podem ser facilmente adicionados usando a API tf.data.

```
def prepare_for_training(ds, cache=True, shuffle_buffer_size=1000):
  # This is a small dataset, only load it once, and keep it in memory.
  # use `.cache(filename)` to cache preprocessing work for datasets that don't
  # fit in memory.
  if cache:
    if isinstance(cache, str):
      ds = ds.cache(cache)
    else:
      ds = ds.cache()

  ds = ds.shuffle(buffer_size=shuffle_buffer_size)

  # Repeat forever
  ds = ds.repeat()

  ds = ds.batch(BATCH_SIZE)

  # `prefetch` lets the dataset fetch batches in the background while the model
  # is training.
  ds = ds.prefetch(buffer_size=AUTOTUNE)

  return ds
```

```
train_ds = prepare_for_training(labeled_ds)

image_batch, label_batch = next(iter(train_ds))
```

```
show_batch(image_batch.numpy(), label_batch.numpy())
```

Performance:

Para investigar, primeiro aqui está uma função para verificar o desempenho de nossos conjuntos de dados:

```
import time
default_timeit_steps = 1000

def timeit(ds, steps=default_timeit_steps):
  start = time.time()
  it = iter(ds)
  for i in range(steps):
    batch = next(it)
    if i%10 == 0:
      print('.',end='')
  print()
  end = time.time()

  duration = end-start
  print("{} batches: {} s".format(steps, duration))
  print("{:0.5f} Images/s".format(BATCH_SIZE*steps/duration))
```

Vamos comparar a velocidade dos dois geradores de dados:

```
# `keras.preprocessing`
timeit(train_data_gen)
```

1000 batches: 76.91638088226318 s

416.03622 Images/s

```
# `tf.data`
timeit(train_ds)
```

1000 batches: 5.537988185882568 s

5778.27163 Images/s

Grande parte do ganho de desempenho vem do uso de .cache.

```
uncached_ds = prepare_for_training(labeled_ds, cache=False)
timeit(uncached_ds)
```

1000 batches: 20.565682888031006 s

1555.99015 Images/s

Se o conjunto de dados não couber na memória, use um arquivo de cache para manter algumas das vantagens:

```
filecache_ds = prepare_for_training(labeled_ds, cache="./flowers.tfcache")
timeit(filecache_ds)
```

1000 batches: 14.327738761901855 s

2233.42989 Images/s