Tutorial

TensorFlow Lite

Esteban Saúl Elizondo Porras

Para comenzar el tutorial tenemos que dirigirnos al siguiente enlace https://colab.research.google.com/github/frogermcs/TFLite-Tester/blob/master/notebooks/Testing_TFLite_model.ipynb

Hay que iniciar sesión con una cuenta de Google para poder proceder al tutorial.

Una vez dentro vamos a ir corriendo y explicando el código sección por sección.

Aquí lo que hacemos es instalar la versión 2.0.0 de tensorflow así como la librería de tensorflow_hub.

```
[ ]  !pip install tensorflow-gpu==2.0.0
     !pip install tensorflow_hub
```

En esta sección importamos todas las librerías necesarias para ejecutar tensorflow.

```
[ ]  from __future__ import absolute_import, division, print_function, unicode_literals

     import matplotlib.pylab as plt
     import tensorflow as tf
     import tensorflow_hub as hub
     import numpy as np
```

En este paso importamos panda para una mejor visualización de los datos e imprimimos las versiones de lo anteriormente instalado.

```
[ ]  import pandas as pd

     # Increase precision of presented data for better side-by-side comparison
     pd.set_option("display.precision", 8)
```

```
[ ]  print("Version: ", tf.__version__)
     print("Hub version: ", hub.__version__)
     print("Eager mode: ", tf.executing_eagerly())
     print("GPU is", "available" if tf.test.is_gpu_available() else "NOT AVAILABLE")
```

Cargamos un paquete de imágenes del enlace.

```
[ ]  data_root = tf.keras.utils.get_file(
       'flower_photos','https://storage.googleapis.com/download.tensorflow.org/example_images/flower_photos.tgz',
        untar=True)
```

Ajustamos el tamaño de la imagen en pixeles, luego ponemos la dirección del directorio donde lo vamos a entrenar.

Luego re escalamos la imagen para ponerla en blanco y negro y la dividimos un 20% para entrenar y validar.

Por último, creamos los generadores de validación y entrenamiento.

```
# Create data generator for training and validation

IMAGE_SHAPE = (224, 224)
TRAINING_DATA_DIR = str(data_root)

datagen_kwargs = dict(rescale=1./255, validation_split=.20)
valid_datagen = tf.keras.preprocessing.image.ImageDataGenerator(**datagen_kwargs)
valid_generator = valid_datagen.flow_from_directory(
    TRAINING_DATA_DIR,
    subset="validation",
    shuffle=True,
    target_size=IMAGE_SHAPE
)

train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(**datagen_kwargs)
train_generator = train_datagen.flow_from_directory(
    TRAINING_DATA_DIR,
    subset="training",
    shuffle=True,
    target_size=IMAGE_SHAPE)
```

Estamos imprimiendo la información que hay dentro del generador de entrenamiento y lo que nos quiere decir es que hay 32 imágenes por 224x224 pixeles y el 3 significa los 3 canales de la escala de colores RGB. El label batch significa que hay 32 etiquetas y 5 clases.

```
# Learn more about data batches

image_batch_train, label_batch_train = next(iter(train_generator))
print("Image batch shape: ", image_batch_train.shape)
print("Label batch shape: ", label_batch_train.shape)
```

```
Image batch shape:  (32, 224, 224, 3)
Label batch shape:  (32, 5)
```

Ordenamos las etiquetas por orden alfabético.

```
# Learn about dataset labels

dataset_labels = sorted(train_generator.class_indices.items(), key=lambda pair:pair[1])
dataset_labels = np.array([key.title() for key, value in dataset_labels])
print(dataset_labels)
```

Vamos a cargar un modelo de preentrenamiento con ciertas características.

Va a funcionar incluso si el paquete de datos tiene un numero diferente de clases

```
model = tf.keras.Sequential([
  hub.KerasLayer("https://tfhub.dev/google/imagenet/mobilenet_v2_100_224/feature_vector/4",
                 output_shape=[1280],
                 trainable=False),
  tf.keras.layers.Dropout(0.4),
  tf.keras.layers.Dense(train_generator.num_classes, activation='softmax')
])
model.build([None, 224, 224, 3])

model.summary()
```

Ya tenemos el modelo, ahora vamos a compilarlo. Se utiliza el algoritmo de Adam para optimizarlo.

```
[ ]  model.compile(
       optimizer=tf.keras.optimizers.Adam(),
       loss='categorical_crossentropy',
       metrics=['acc'])
```

Vamos a entrenar nuestro algoritmo con 10 epochs, entre más epochs tenga mejor se va a entrenar, pero va a durar más.

```
steps_per_epoch = np.ceil(train_generator.samples/train_generator.batch_size)
val_steps_per_epoch = np.ceil(valid_generator.samples/valid_generator.batch_size)

hist = model.fit(
    train_generator,
    epochs=10,
    verbose=1,
    steps_per_epoch=steps_per_epoch,
    validation_data=valid_generator,
    validation_steps=val_steps_per_epoch).history
```

Después de entrenado vamos a plotear los gráficos de pérdidas y precisión para el entrenamiento y la validación.

```
plt.figure()
plt.ylabel("Loss (training and validation)")
plt.xlabel("Training Steps")
plt.ylim([0,2])
plt.plot(hist["loss"])
plt.plot(hist["val_loss"])

plt.figure()
plt.ylabel("Accuracy (training and validation)")
plt.xlabel("Training Steps")
plt.ylim([0,1])
plt.plot(hist["acc"])
plt.plot(hist["val_acc"])
```

Ahora debemos guardar el modelo en formato de TensorFlow para que sea exportado.

```
FLOWERS_SAVED_MODEL = "saved_models/flowers3"
tf.saved_model.save(model, FLOWERS_SAVED_MODEL)
```

Cargamos el modelo previamente guardado.

```
# Load SavedModel

flowers_model = hub.load(FLOWERS_SAVED_MODEL)
print(flowers_model)
```

Chequeamos la validación del paquete.

```
# Get images and labels batch from validation dataset generator

val_image_batch, val_label_batch = next(iter(valid_generator))
true_label_ids = np.argmax(val_label_batch, axis=-1)

print("Validation batch shape:", val_image_batch.shape)
```

```
Validation batch shape: (32, 224, 224, 3)
```

Y las predicciones.

```
tf_model_predictions = flowers_model(val_image_batch)
print("Prediction results shape:", tf_model_predictions.shape)
```

```
Prediction results shape: (32, 5)
```

Pasamos las predicciones al formato de panda para una mejor visualización y lo imprimimos.

```
tf_pred_dataframe = pd.DataFrame(tf_model_predictions.numpy())
tf_pred_dataframe.columns = dataset_labels

print("Prediction results for the first elements")
tf_pred_dataframe.head()
```

Se emparejan los id de las predicciones con las etiquetas y se plotea el resultado.

```
predicted_ids = np.argmax(tf_model_predictions, axis=-1)
predicted_labels = dataset_labels[predicted_ids]
```

```
# Print images batch and labels predictions

plt.figure(figsize=(10,9))
plt.subplots_adjust(hspace=0.5)
for n in range(30):
  plt.subplot(6,5,n+1)
  plt.imshow(val_image_batch[n])
  color = "green" if predicted_ids[n] == true_label_ids[n] else "red"
  plt.title(predicted_labels[n].title(), color=color)
  plt.axis('off')
_ = plt.suptitle("Model predictions (green: correct, red: incorrect)")
```

Ahora toca convertirlo a tflite y a un modelo aun más optimizado de tflite llamado quantized.

```
TFLITE_MODEL = "tflite_models/flowers.tflite"
TFLITE_QUANT_MODEL = "tflite_models/flowers_quant.tflite"
```

```
# Get the concrete function from the Keras model.
run_model = tf.function(lambda x : flowers_model(x))

# Save the concrete function.
concrete_func = run_model.get_concrete_function(
    tf.TensorSpec(model.inputs[0].shape, model.inputs[0].dtype)
)

# Convert the model
converter = tf.lite.TFLiteConverter.from_concrete_functions([concrete_func])
converted_tflite_model = converter.convert()
open(TFLITE_MODEL, "wb").write(converted_tflite_model)

# Convert the model to quantized version with post-training quantization
converter = tf.lite.TFLiteConverter.from_concrete_functions([concrete_func])
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
tflite_quant_model = converter.convert()
open(TFLITE_QUANT_MODEL, "wb").write(tflite_quant_model)

print("TFLite models and their sizes:")
!ls "tflite_models" -lh
```

```
TFLite models and their sizes:
total 11M
-rw-r--r-- 1 root root 2.3M Oct 27 11:15 flowers_quant.tflite
-rw-r--r-- 1 root root 8.5M Oct 27 11:15 flowers.tflite
```

Cargamos el modelo a tflite e imprimimos su información de entrada y salida.

```
# Load TFLite model and see some details about input/output

tflite_interpreter = tf.lite.Interpreter(model_path=TFLITE_MODEL)

input_details = tflite_interpreter.get_input_details()
output_details = tflite_interpreter.get_output_details()

print("== Input details ==")
print("name:", input_details[0]['name'])
print("shape:", input_details[0]['shape'])
print("type:", input_details[0]['dtype'])

print("\n== Output details ==")
print("name:", output_details[0]['name'])
print("shape:", output_details[0]['shape'])
print("type:", output_details[0]['dtype'])
```

```
== Input details ==
name: x
shape: [  1 224 224   3]
type: <class 'numpy.float32'>

== Output details ==
name: Identity
shape: [1 5]
type: <class 'numpy.float32'>
```

Editamos el tamaño de los tensores para hacer predicciones para grupos de 32.

```
tflite_interpreter.resize_tensor_input(input_details[0]['index'], (32, 224, 224, 3))
tflite_interpreter.resize_tensor_input(output_details[0]['index'], (32, 5))
tflite_interpreter.allocate_tensors()

input_details = tflite_interpreter.get_input_details()
output_details = tflite_interpreter.get_output_details()

print("== Input details ==")
print("name:", input_details[0]['name'])
print("shape:", input_details[0]['shape'])
print("type:", input_details[0]['dtype'])

print("\n== Output details ==")
print("name:", output_details[0]['name'])
print("shape:", output_details[0]['shape'])
print("type:", output_details[0]['dtype'])
```

```
== Input details ==
name: x
shape: [ 32 224 224   3]
type: <class 'numpy.float32'>

== Output details ==
name: Identity
shape: [32  5]
type: <class 'numpy.float32'>
```

Ejecutamos el interpreter para ver los resultados de la predicción.

```
tflite_interpreter.set_tensor(input_details[0]['index'], val_image_batch)

tflite_interpreter.invoke()

tflite_model_predictions = tflite_interpreter.get_tensor(output_details[0]['index'])
print("Prediction results shape:", tflite_model_predictions.shape)
```

```
Prediction results shape: (32, 5)
```

Realizamos el mismo paso pero para el modelo quantized.

```
# Load quantized TFLite model
tflite_interpreter_quant = tf.lite.Interpreter(model_path=TFLITE_QUANT_MODEL)

# Learn about its input and output details
input_details = tflite_interpreter_quant.get_input_details()
output_details = tflite_interpreter_quant.get_output_details()

# Resize input and output tensors to handle batch of 32 images
tflite_interpreter_quant.resize_tensor_input(input_details[0]['index'], (32, 224, 224, 3))
tflite_interpreter_quant.resize_tensor_input(output_details[0]['index'], (32, 5))
tflite_interpreter_quant.allocate_tensors()

input_details = tflite_interpreter_quant.get_input_details()
output_details = tflite_interpreter_quant.get_output_details()

print("== Input details ==")
print("name:", input_details[0]['name'])
print("shape:", input_details[0]['shape'])
print("type:", input_details[0]['dtype'])

print("\n== Output details ==")
print("name:", output_details[0]['name'])
print("shape:", output_details[0]['shape'])
print("type:", output_details[0]['dtype'])

# Run inference
tflite_interpreter_quant.set_tensor(input_details[0]['index'], val_image_batch)

tflite_interpreter_quant.invoke()

tflite_q_model_predictions = tflite_interpreter_quant.get_tensor(output_details[0]['index'])
print("\nPrediction results shape:", tflite_q_model_predictions.shape)
```

Una vez tabulados todos los resultados podemos ver la comparación de los 3 métodos.

	Daisy		
	TF Model	TFLite	TFLite quantized
0	0.00022492342	0.00022492348	4.9308896e-06
1	0.0015232594	0.0015232647	0.0006356425
2	1.1647387e-05	1.1647244e-05	8.7856733e-06
3	1.4158371e-05	1.4158586e-05	0.0015058323
4	0.93151599	0.93151623	0.97616559
5	0.153574	0.15357304	0.47339907
6	7.5707128e-08	7.5706126e-08	6.3441252e-07
7	0.00039845117	0.00039844928	0.00018309847
8	0.0019768912	0.001976914	0.0015610195
9	9.8064049e-08	9.8061811e-08	4.8461608e-09
10	0.00032995452	0.00032995269	0.030962177
11	0.0019018369	0.0019018062	0.013192666
12	3.2076856e-05	3.2077187e-05	0.00039367317
13	2.7161666e-05	2.7161406e-05	1.0562387e-06
14	0.0002821784	0.00028217977	0.00034582219
15	0.0048273397	0.0048272782	0.0043780231
16	0.0025830294	0.0025830609	0.010829359
17	7.6628807e-05	7.6628159e-05	0.0003728102
18	9.058811e-05	9.058787e-05	4.809429e-07
19	0.99866116	0.99866116	0.37706217
20	0.96293551	0.96293533	0.98912197
21	0.0010520749	0.0010520765	0.02279035
22	1.8794697e-05	1.8794446e-05	0.0002823096
23	0.0075597526	0.0075598261	0.030620387
24	0.0003092156	0.00030921435	8.367886e-06
25	0.00059458578	0.00059458357	1.1506654e-05

Como podemos ver el TF y TFLite difieren por muy poco y en algunos casos es igual pero como el quantized TFLite si se nota mucho más la diferencia pero también hay que ver que pesa como 3 o 4 veces menos que el de TFLite.