index

October 27, 2021

1 Train Your Own Model and Convert It to TFLite (uncompleted work)

/ This notebook uses the Fashion MNIST dataset which contains 70,000 grayscale images in 10 categories. The images show individual articles of clothing at low resolution (28 by 28 pixels), as seen here:

Figure 1. Fashion-MNIST samples (by Zalando, MIT License).

Fashion MNIST is intended as a drop-in replacement for the classic MNIST dataset—often used as the “Hello, World” of machine learning programs for computer vision. The MNIST dataset contains images of handwritten digits (0, 1, 2, etc.) in a format identical to that of the articles of clothing we’ll use here.

This uses Fashion MNIST for variety, and because it’s a slightly more challenging problem than regular MNIST. Both datasets are relatively small and are used to verify that an algorithm works as expected. They’re good starting points to test and debug code.

We will use 60,000 images to train the network and 10,000 images to evaluate how accurately the network learned to classify images. You can access the Fashion MNIST directly from TensorFlow. Import and load the Fashion MNIST data directly from TensorFlow:

[1]:
```
try:
    %tensorflow_version 2.x
except:
    pass
```

```
TensorFlow 2.x selected.
```

2 Setup

[2]:
```
# TensorFlow
import tensorflow as tf

# TensorFlow Datsets
import tensorflow_datasets as tfds
tfds.disable_progress_bar()
```

```python
# Helper Libraries
import numpy as np
import matplotlib.pyplot as plt
import pathlib

from os import getcwd

print('\u2022 Using TensorFlow Version:', tf.__version__)
print('\u2022 GPU Device Found.' if tf.test.is_gpu_available() else '\u2022 GPU␣
↪Device Not Found. Running on CPU')
```

```
• Using TensorFlow Version: 2.1.0-rc1
WARNING:tensorflow:From <ipython-input-2-bc076dfff1bf>:15: is_gpu_available
(from tensorflow.python.framework.test_util) is deprecated and will be removed
in a future version.
Instructions for updating:
Use `tf.config.list_physical_devices('GPU')` instead.
• GPU Device Found.
```

3 Download Fashion MNIST Dataset

We will use TensorFlow Datasets to load the Fashion MNIST dataset.

[3]:
```python
splits = tfds.Split.ALL.subsplit(weighted=(80, 10, 10))

filePath = f"{getcwd()}/../tmp2/"
splits, info = tfds.load('fashion_mnist', with_info=True, as_supervised=True,␣
↪split=splits, data_dir=filePath)

(train_examples, validation_examples, test_examples) = splits

num_examples = info.splits['train'].num_examples
num_classes = info.features['label'].num_classes
```

Downloading and preparing dataset fashion_mnist (29.45 MiB) to

/content/../tmp2/fashion_mnist/1.0.0…

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/tensorflow_datasets/core/file_format_adapter.py:209: tf_record_iterator
(from tensorflow.python.lib.io.tf_record) is deprecated and will be removed in a
future version.
Instructions for updating:
Use eager execution and:
`tf.data.TFRecordDataset(path)`
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/tensorflow_datasets/core/file_format_adapter.py:209: tf_record_iterator
(from tensorflow.python.lib.io.tf_record) is deprecated and will be removed in a
```

```
future version.
Instructions for updating:
Use eager execution and:
`tf.data.TFRecordDataset(path)`
```

Dataset fashion_mnist downloaded and prepared to /content/../tmp2/fashion_mnist/1.0.0. Subsequent calls will reuse this data.

The class names are not included with the dataset, so we will specify them here.

[0]:
```
class_names = ['T-shirt_top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

[0]:
```
# Create a labels.txt file with the class names
with open('labels.txt', 'w') as f:
    f.write('\n'.join(class_names))
```

[0]:
```
# The images in the dataset are 28 by 28 pixels.
pixels = 28
IMG_SIZE = (pixels, pixels)
```

4 Preprocessing Data

4.1 Preprocess

[0]:
```
# EXERCISE: Write a function to normalize the images.

def format_example(image, label):
    # Cast image to float32
    image = tf.image.convert_image_dtype(image, tf.float32)

    # Normalize the image in the range [0, 1]
    image = tf.image.resize(image, IMG_SIZE) / 255.0

    return image, label
```

[0]:
```
# Specify the batch size
BATCH_SIZE = 256
```

4.2 Create Datasets From Images and Labels

[0]:
```
# Create Datasets
train_batches = train_examples.cache().shuffle(num_examples//4).
 ↪batch(BATCH_SIZE).map(format_example).prefetch(1)
validation_batches = validation_examples.cache().batch(BATCH_SIZE).
 ↪map(format_example)
test_batches = test_examples.map(format_example).batch(1)
```

5 Building the Model

```
Model: "sequential"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 16)        160
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 32)        4640
_________________________________________________________________
flatten (Flatten)            (None, 3872)              0
_________________________________________________________________
dense (Dense)                (None, 64)                247872
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 253,322
Trainable params: 253,322
Non-trainable params: 0
```

[0]:

```
# EXERCISE: Build and compile the model shown in the previous cell.

model = tf.keras.Sequential([
    # Set the input shape to (28, 28, 1), kernel size=3, filters=16 and use␣
 ↪ReLU activation,
    tf.keras.layers.Conv2D(filters=16, kernel_size=3,activation='relu'),

    tf.keras.layers.MaxPooling2D(),

    # Set the number of filters to 32, kernel size to 3 and use ReLU activation
    tf.keras.layers.Conv2D(filters=16,kernel_size=3,activation='relu'),

    # Flatten the output layer to 1 dimension
    tf.keras.layers.Flatten(),

    # Add a fully connected layer with 64 hidden units and ReLU activation
    tf.keras.layers.Dense(units=64,activation='relu'),

    # Attach a final softmax classification head
    tf.keras.layers.Dense(units=64,activation='softmax')])

# Set the appropriate loss function and use accuracy as your metric
model.compile(optimizer='adam',
              loss= 'sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

5.1 Train

[11]:
```
history = model.fit(train_batches, epochs=10,␣
    ↪validation_data=validation_batches)
```

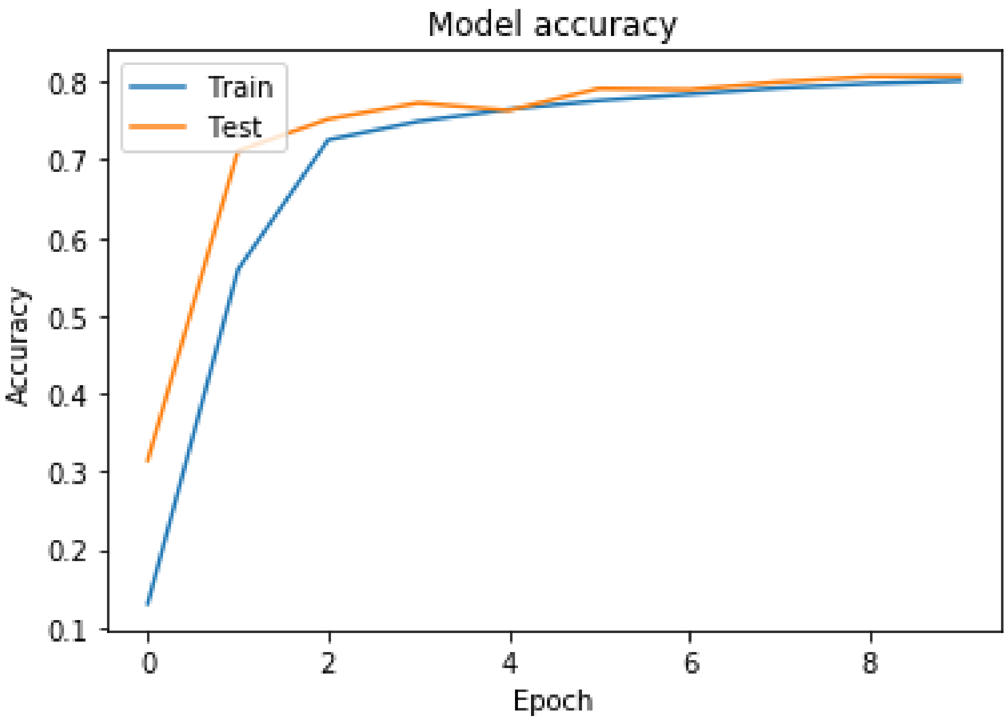
```
Epoch 1/10
219/219 [==============================] - 16s 75ms/step - loss: 2.4445 -
accuracy: 0.1303 - val_loss: 2.2006 - val_accuracy: 0.3146
Epoch 2/10
219/219 [==============================] - 1s 6ms/step - loss: 1.3332 -
accuracy: 0.5592 - val_loss: 0.8000 - val_accuracy: 0.7116
Epoch 3/10
219/219 [==============================] - 1s 6ms/step - loss: 0.7371 -
accuracy: 0.7257 - val_loss: 0.6831 - val_accuracy: 0.7524
Epoch 4/10
219/219 [==============================] - 1s 6ms/step - loss: 0.6662 -
accuracy: 0.7495 - val_loss: 0.6328 - val_accuracy: 0.7733
Epoch 5/10
219/219 [==============================] - 1s 6ms/step - loss: 0.6277 -
accuracy: 0.7649 - val_loss: 0.6157 - val_accuracy: 0.7636
Epoch 6/10
219/219 [==============================] - 1s 6ms/step - loss: 0.6027 -
accuracy: 0.7763 - val_loss: 0.5776 - val_accuracy: 0.7917
Epoch 7/10
219/219 [==============================] - 1s 6ms/step - loss: 0.5832 -
accuracy: 0.7844 - val_loss: 0.5622 - val_accuracy: 0.7903
Epoch 8/10
219/219 [==============================] - 1s 6ms/step - loss: 0.5656 -
accuracy: 0.7924 - val_loss: 0.5480 - val_accuracy: 0.7999
Epoch 9/10
219/219 [==============================] - 1s 6ms/step - loss: 0.5525 -
accuracy: 0.7977 - val_loss: 0.5416 - val_accuracy: 0.8067
Epoch 10/10
219/219 [==============================] - 1s 5ms/step - loss: 0.5431 -
accuracy: 0.8014 - val_loss: 0.5258 - val_accuracy: 0.8070
```

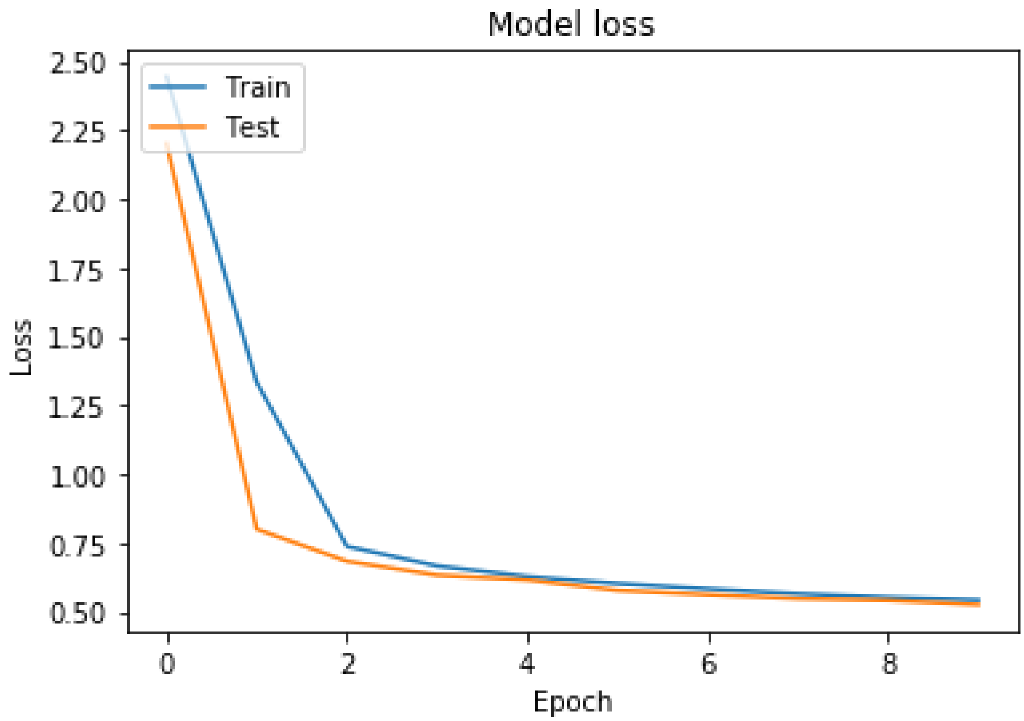
[16]:
```
history_dict = history.history
print(history_dict.keys())
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

[17]:
```
# Plot training & validation accuracy values
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

```
# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```





6 Exporting to TFLite

You will now save the model to TFLite. We should note, that you will probably see some warning messages when running the code below. These warnings have to do with software updates and should not cause any errors or prevent your code from running.

[0]:
```
# EXERCISE: Use the tf.saved_model API to save your model in the SavedModel␣
↪format.
export_dir = 'saved_model/1'

# YOUR CODE HERE
RPS_SAVED_MODEL = "rps_saved_model"
tf.saved_model.save(model, RPS_SAVED_MODEL)
loaded = tf.saved_model.load(RPS_SAVED_MODEL)
print(list(loaded.signatures.keys()))
infer = loaded.signatures["serving_default"]
print(infer.structured_input_signature)
print(infer.structured_outputs)
```

```
INFO:tensorflow:Assets written to: rps_saved_model/assets
```

```
INFO:tensorflow:Assets written to: rps_saved_model/assets
```

```
['serving_default']
((), {'input_1': TensorSpec(shape=(None, 28, 28, 1), dtype=tf.float32,
```

```
name='input_1')})
{'output_1': TensorSpec(shape=(None, 64), dtype=tf.float32, name='output_1')}
```

[0]:
```
# Select mode of optimization
mode = "Speed"

if mode == 'Storage':
    optimization = tf.lite.Optimize.OPTIMIZE_FOR_SIZE
elif mode == 'Speed':
    optimization = tf.lite.Optimize.OPTIMIZE_FOR_LATENCY
else:
    optimization = tf.lite.Optimize.DEFAULT
```

[0]:
```
# EXERCISE: Use the TFLiteConverter SavedModel API to initialize the converter

converter = tf.lite.TFLiteConverter.from_saved_model(RPS_SAVED_MODEL)
# Set the optimzations
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]

# Invoke the converter to finally generate the TFLite model
tflite_model = converter.convert()
```

[0]:
```
tflite_model_file = pathlib.Path('./model.tflite')
tflite_model_file.write_bytes(tflite_model)
```

[0]:
```
134224
```

7 Test the Model with TFLite Interpreter

---

[0]:
```
# Load TFLite model and allocate tensors.
interpreter = tf.lite.Interpreter(model_content=tflite_model)
interpreter.allocate_tensors()

input_index = interpreter.get_input_details()[0]["index"]
output_index = interpreter.get_output_details()[0]["index"]
```

[0]:
```
# Gather results for the randomly sampled test images
predictions = []
test_labels = []
test_images = []

for img, label in test_batches.take(50):
    interpreter.set_tensor(input_index, img)
    interpreter.invoke()
    predictions.append(interpreter.get_tensor(output_index))
```

```
    test_labels.append(label[0])
    test_images.append(np.array(img))
```

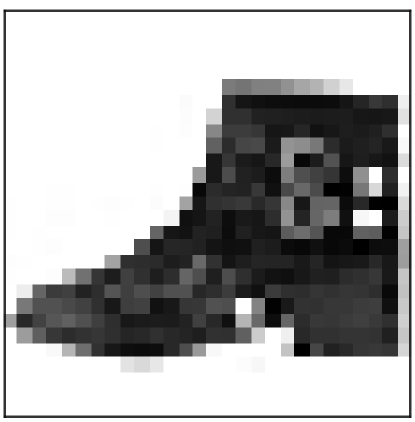
[0]:
```
# Utilities functions for plotting

def plot_image(i, predictions_array, true_label, img):
    predictions_array, true_label, img = predictions_array[i], true_label[i],␣
 ↪img[i]
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])

    img = np.squeeze(img)

    plt.imshow(img, cmap=plt.cm.binary)

    predicted_label = np.argmax(predictions_array)

    if predicted_label == true_label.numpy():
        color = 'green'
    else:
        color = 'red'

    plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                          100*np.max(predictions_array),
                                          class_names[true_label]),
                                          color=color)

def plot_value_array(i, predictions_array, true_label):
    predictions_array, true_label = predictions_array[i], true_label[i]
    plt.grid(False)
    plt.xticks(list(range(10)))
    plt.yticks([])
    thisplot = plt.bar(range(10), predictions_array[0], color="#777777")
    plt.ylim([0, 1])
    predicted_label = np.argmax(predictions_array[0])

    thisplot[predicted_label].set_color('red')
    thisplot[true_label].set_color('blue')
```

[0]:
```
# Visualize the outputs

# Select index of image to display. Minimum index value is 1 and max index␣
 ↪value is 50.
index = 49

plt.figure(figsize=(6,3))
```

```
plt.subplot(1,2,1)
plot_image(index, predictions, test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(index, predictions, test_labels)
plt.show()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-44-d790dfad8f59> in <module>()
      5 plot_image(index, predictions, test_labels, test_images)
      6 plt.subplot(1,2,2)
----> 7 plot_value_array(index, predictions, test_labels)
      8 plt.show()

<ipython-input-43-422f24660cf3> in plot_value_array(i, predictions_array,␣
 ↪true_label)
     27     plt.xticks(list(range(10)))
     28     plt.yticks([])
---> 29     thisplot = plt.bar(range(10), predictions_array[0], color="#777777"
     30     plt.ylim([0, 1])
     31     predicted_label = np.argmax(predictions_array[0])

/usr/local/lib/python3.6/dist-packages/matplotlib/pyplot.py in bar(x, height,␣
 ↪width, bottom, align, data, **kwargs)
   2432     return gca().bar(
   2433         x, height, width=width, bottom=bottom, align=align,
-> 2434         **({"data": data} if data is not None else {}), **kwargs)

   2435
   2436

/usr/local/lib/python3.6/dist-packages/matplotlib/__init__.py in inner(ax, data ␣
 ↪*args, **kwargs)
   1597     def inner(ax, *args, data=None, **kwargs):
   1598         if data is None:
-> 1599             return func(ax, *map(sanitize_sequence, args), **kwargs)
   1600
   1601         bound = new_sig.bind(ax, *args, **kwargs)

/usr/local/lib/python3.6/dist-packages/matplotlib/axes/_axes.py in bar(self, x,␣
 ↪height, width, bottom, align, **kwargs)
   2372         x, height, width, y, linewidth = np.broadcast_arrays(
   2373             # Make args iterable too.
-> 2374             np.atleast_1d(x), height, width, y, linewidth)

   2375
   2376         # Now that units have been converted, set the tick locations.
```

```
<__array_function__ internals> in broadcast_arrays(*args, **kwargs)

/usr/local/lib/python3.6/dist-packages/numpy/lib/stride_tricks.py in broadcast_arrays(*args, **kwargs)
    262     args = [np.array(_m, copy=False, subok=subok) for _m in args]
    263
--> 264     shape = _broadcast_shape(*args)
    265
    266     if all(array.shape == shape for array in args):

/usr/local/lib/python3.6/dist-packages/numpy/lib/stride_tricks.py in _broadcast_shape(*args)
    189     # use the old-iterator because np.nditer does not handle size 0 arrays
    190     # consistently
--> 191     b = np.broadcast(*args[:32])
    192     # unfortunately, it cannot handle 32 or more arguments directly
    193     for pos in range(32, len(args), 31):

ValueError: shape mismatch: objects cannot be broadcast to a single shape
```


Ankle boot 99% (Ankle boot)