Open in Colab

Ungraded Lab: Quantization and Pruning

In this lab, you will get some hands-on practice with the mobile optimization techniques discussed in the lectures. These enable reduced model size and latency which makes it ideal for edge and IOT devices. You will start by training a Keras model then compare its model size and accuracy after going through these techniques:

- post-training quantization
- quantization aware training
- weight pruning

Let's begin!

Imports

Let's first import a few common libraries that you'll be using throughout the notebook.

In [1]:
```
import tensorflow as tf
import numpy as np
import os
import tempfile
import zipfile
```

Utilities and constants

Let's first define a few string constants and utility functions to make our code easier to maintain.

In [2]:
```
# GLOBAL VARIABLES

# String constants for model filenames
FILE_WEIGHTS = 'baseline_weights.h5'
FILE_NON_QUANTIZED_H5 = 'non_quantized.h5'
FILE_NON_QUANTIZED_TFLITE = 'non_quantized.tflite'
FILE_PT_QUANTIZED = 'post_training_quantized.tflite'
FILE_QAT_QUANTIZED = 'quant_aware_quantized.tflite'
FILE_PRUNED_MODEL_H5 = 'pruned_model.h5'
FILE_PRUNED_QUANTIZED_TFLITE = 'pruned_quantized.tflite'
FILE_PRUNED_NON_QUANTIZED_TFLITE = 'pruned_non_quantized.tflite'

# Dictionaries to hold measurements
MODEL_SIZE = {}
ACCURACY = {}
```

In [3]:

```
# UTILITY FUNCTIONS

def print_metric(metric_dict, metric_name):
  '''Prints key and values stored in a dictionary'''
  for metric, value in metric_dict.items():
    print(f'{metric_name} for {metric}: {value}')


def model_builder():
  '''Returns a shallow CNN for training on the MNIST dataset'''

  keras = tf.keras

  # Define the model architecture.
  model = keras.Sequential([
    keras.layers.InputLayer(input_shape=(28, 28)),
    keras.layers.Reshape(target_shape=(28, 28, 1)),
    keras.layers.Conv2D(filters=12, kernel_size=(3, 3), activation='relu'),
    keras.layers.MaxPooling2D(pool_size=(2, 2)),
    keras.layers.Flatten(),
    keras.layers.Dense(10, activation='softmax')
  ])

  return model


def evaluate_tflite_model(filename, x_test, y_test):
  '''
  Measures the accuracy of a given TF Lite model and test set

  Args:
    filename (string) - filename of the model to load
    x_test (numpy array) - test images
    y_test (numpy array) - test labels

  Returns
    float showing the accuracy against the test set
  '''

  # Initialize the TF Lite Interpreter and allocate tensors
  interpreter = tf.lite.Interpreter(model_path=filename)
  interpreter.allocate_tensors()

  # Get input and output index
  input_index = interpreter.get_input_details()[0]["index"]
  output_index = interpreter.get_output_details()[0]["index"]

  # Initialize empty predictions list
  prediction_digits = []

  # Run predictions on every image in the "test" dataset.
  for i, test_image in enumerate(x_test):
    # Pre-processing: add batch dimension and convert to float32 to match with
    # the model's input data format.
    test_image = np.expand_dims(test_image, axis=0).astype(np.float32)
    interpreter.set_tensor(input_index, test_image)

    # Run inference.
    interpreter.invoke()

    # Post-processing: remove batch dimension and find the digit with highest
    # probability.
    output = interpreter.tensor(output_index)
    digit = np.argmax(output()[0])
    prediction_digits.append(digit)

  # Compare prediction results with ground truth labels to calculate accuracy.
  prediction_digits = np.array(prediction_digits)
  accuracy = (prediction_digits == y_test).mean()

  return accuracy


def get_gzipped_model_size(file):
  '''Returns size of gzipped model, in bytes.'''
  _, zipped_file = tempfile.mkstemp('.zip')
  with zipfile.ZipFile(zipped_file, 'w', compression=zipfile.ZIP_DEFLATED) as f:
    f.write(file)

  return os.path.getsize(zipped_file)
```

Download and Prepare the Dataset

You will be using the MNIST dataset which is hosted in Keras Datasets. Some of the helper files in this notebook are made to work with this dataset so if you decide to switch to a different dataset, make sure to check if those helper functions need to be modified (e.g. shape of the Flatten layer in your model).

In [4]:
```
# Load MNIST dataset
mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Normalize the input image so that each pixel value is between 0 to 1.
train_images = train_images / 255.0
test_images = test_images / 255.0
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
```

Baseline Model

You will first build and train a Keras model. This will be the baseline where you will be comparing the mobile optimized versions later on. This will just be a shallow CNN with a softmax output to classify a given MNIST digit. You can review the `model_builder()` function in the utilities at the top of this notebook but we also printed the model summary below to show the architecture.

You will also save the weights so you can reinitialize the other models later the same way. This is not needed in real projects but for this demo notebook, it would be good to have the same initial state later so you can compare the effects of the optimizations.

In [5]:
```
# Create the baseline model
baseline_model = model_builder()

# Save the initial weights for use later
baseline_model.save_weights(FILE_WEIGHTS)

# Print the model summary
baseline_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
reshape (Reshape)            (None, 28, 28, 1)         0
_________________________________________________________________
conv2d (Conv2D)              (None, 26, 26, 12)        120
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 12)        0
_________________________________________________________________
flatten (Flatten)            (None, 2028)              0
_________________________________________________________________
dense (Dense)                (None, 10)                20290
=================================================================
Total params: 20,410
Trainable params: 20,410
Non-trainable params: 0
_________________________________________________________________
```

You can then compile and train the model. In practice, it's best to shuffle the train set but for this demo, it is set to `False` for reproducibility of the results. One epoch below will reach around 91% accuracy.

In [6]:
```
# Setup the model for training
baseline_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
baseline_model.fit(train_images, train_labels, epochs=1, shuffle=False)
```

```
1875/1875 [==============================] - 21s 11ms/step - loss: 0.2975 - accuracy: 0.9172
```

Out[6]:
```
<tensorflow.python.keras.callbacks.History at 0x7eff96913350>
```

Let's save the accuracy of the model against the test set so you can compare later.

In [7]:
```
# Get the baseline accuracy
_, ACCURACY['baseline Keras model'] = baseline_model.evaluate(test_images, test_labels)
```

```
313/313 [==============================] - 2s 6ms/step - loss: 0.1457 - accuracy: 0.9574
```

Next, you will save the Keras model as a file and record its size as well.

In [8]:
```
# Save the Keras model
baseline_model.save(FILE_NON_QUANTIZED_H5, include_optimizer=False)

# Save and get the model size
MODEL_SIZE['baseline h5'] = os.path.getsize(FILE_NON_QUANTIZED_H5)

# Print records so far
print_metric(ACCURACY, "test accuracy")
print_metric(MODEL_SIZE, "model size in bytes")
```

```
test accuracy for baseline Keras model: 0.9574000239372253
model size in bytes for baseline h5: 98136
```

Convert the model to TF Lite format

Next, you will convert the model to Tensorflow Lite (TF Lite) format. This is designed to make Tensorflow models more efficient and lightweight when running on mobile, embedded, and IOT devices.

You can convert a Keras model with TF Lite's Converter class and we've incorporated it in the short helper function below. Notice that there is a `quantize` flag which you can use to quantize the model.

In [9]:
```
def convert_tflite(model, filename, quantize=False):
  '''
  Converts the model to TF Lite format and writes to a file

  Args:
    model (Keras model) - model to convert to TF Lite
    filename (string) - string to use when saving the file
    quantize (bool) - flag to indicate quantization

  Returns:
    None
  '''

  # Initialize the converter
  converter = tf.lite.TFLiteConverter.from_keras_model(model)

  # Set for quantization if flag is set to True
  if quantize:
    converter.optimizations = [tf.lite.Optimize.DEFAULT]

  # Convert the model
  tflite_model = converter.convert()

  # Save the model.
  with open(filename, 'wb') as f:
    f.write(tflite_model)
```

You will use the helper function to convert the Keras model then get its size and accuracy. Take note that this is *not yet* quantized.

In [10]:
```
# Convert baseline model
convert_tflite(baseline_model, FILE_NON_QUANTIZED_TFLITE)
```

```
INFO:tensorflow:Assets written to: /tmp/tmpwm9nwfr2/assets
```

You will notice that there is already a slight decrease in model size when converting to `.tflite` format.

In [11]:
```
MODEL_SIZE['non quantized tflite'] = os.path.getsize(FILE_NON_QUANTIZED_TFLITE)

print_metric(MODEL_SIZE, 'model size in bytes')
```

```
model size in bytes for baseline h5: 98136
model size in bytes for non quantized tflite: 84740
```

The accuracy will also be nearly identical when converting between formats. You can setup a TF Lite model for input-output using its Interpreter class. This is shown in the `evaluate_tflite_model()` helper function provided in the `Utilities` section earlier.

Note: If you see a `Runtime Error: There is at least 1 reference to internal data in the interpreter in the form of a numpy array or slice.` , please try re-running the cell.

In [12]:
```
ACCURACY['non quantized tflite'] = evaluate_tflite_model(FILE_NON_QUANTIZED_TFLITE, test_images, test_
```

In [13]:
```
print_metric(ACCURACY, 'test accuracy')
```

```
test accuracy for baseline Keras model: 0.9574000239372253
test accuracy for non quantized tflite: 0.9574
```

Post-Training Quantization

Now that you have the baseline metrics, you can now observe the effects of quantization. As mentioned in the lectures, this process involves converting floating point representations into integer to reduce model size and achieve faster computation.

As shown in the `convert_tflite()` helper function earlier, you can easily do post-training quantization with the TF Lite API. You just need to set the converter optimization and assign an Optimize Enum.

You will set the `quantize` flag to do that and get the metrics again.

In [14]:
```
# Convert and quantize the baseline model
convert_tflite(baseline_model, FILE_PT_QUANTIZED, quantize=True)
```

```
INFO:tensorflow:Assets written to: /tmp/tmpre3nw7_1/assets
INFO:tensorflow:Assets written to: /tmp/tmpre3nw7_1/assets
```

In [15]:
```
# Get the model size
MODEL_SIZE['post training quantized tflite'] = os.path.getsize(FILE_PT_QUANTIZED)

print_metric(MODEL_SIZE, 'model size')
```

```
model size for baseline h5: 98136
model size for non quantized tflite: 84740
model size for post training quantized tflite: 24160
```

You should see around a 4X reduction in model size in the quantized version. This comes from converting the 32 bit representations (float) into 8 bits (integer).

In [16]:
```
ACCURACY['post training quantized tflite'] = evaluate_tflite_model(FILE_PT_QUANTIZED, test_images, te
```

In [17]:
```
print_metric(ACCURACY, 'test accuracy')
```

```
test accuracy for baseline Keras model: 0.9574000239372253
test accuracy for non quantized tflite: 0.9574
test accuracy for post training quantized tflite: 0.9569
```

As mentioned in the lecture, you can expect the accuracy to not be the same when quantizing the model. Most of the time it will decrease but in some cases, it can even increase. Again, this can be attributed to the loss of precision when you remove the extra bits from the float data.

Quantization Aware Training

When post-training quantization results in loss of accuracy that is unacceptable for your application, you can consider doing quantization aware training before quantizing the model. This simulates the loss of precision by inserting fake quant nodes in the model during training. That way, your model will learn to adapt with the loss of precision to get more accurate predictions.

The Tensorflow Model Optimization Toolkit provides a quantize_model() method to do this quickly and you will see that below. But first, let's install the toolkit into the notebook environment.

In [18]:
```
# Install the toolkit
!pip install tensorflow_model_optimization
```

```
Collecting tensorflow_model_optimization
  Downloading tensorflow_model_optimization-0.6.0-py2.py3-none-any.whl (211 kB)
     |████████████████████████████████| 211 kB 5.2 MB/s
Requirement already satisfied: numpy~=1.14 in /usr/local/lib/python3.7/dist-packages (from tensorflow_
model_optimization) (1.19.5)
Requirement already satisfied: dm-tree~=0.1.1 in /usr/local/lib/python3.7/dist-packages (from tensorfl
ow_model_optimization) (0.1.6)
Requirement already satisfied: six~=1.10 in /usr/local/lib/python3.7/dist-packages (from tensorflow_mo
del_optimization) (1.15.0)
Installing collected packages: tensorflow-model-optimization
Successfully installed tensorflow-model-optimization-0.6.0
```

You will build the baseline model again but this time, you will pass it into the `quantize_model()` method to indicate

quantization aware training.

Take note that in case you decide to pass in a model that is already trained, then make sure to recompile before you continue training.

In [19]:
```
import tensorflow_model_optimization as tfmot

# method to quantize a Keras model
quantize_model = tfmot.quantization.keras.quantize_model

# Define the model architecture.
model_to_quantize = model_builder()

# Reinitialize weights with saved file
model_to_quantize.load_weights(FILE_WEIGHTS)

# Quantize the model
q_aware_model = quantize_model(model_to_quantize)

# `quantize_model` requires a recompile.
q_aware_model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

q_aware_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
quantize_layer (QuantizeLaye (None, 28, 28)            3
_________________________________________________________________
quant_reshape_1 (QuantizeWra (None, 28, 28, 1)         1
_________________________________________________________________
quant_conv2d_1 (QuantizeWrap (None, 26, 26, 12)        147
_________________________________________________________________
quant_max_pooling2d_1 (Quant (None, 13, 13, 12)        1
_________________________________________________________________
quant_flatten_1 (QuantizeWra (None, 2028)              1
_________________________________________________________________
quant_dense_1 (QuantizeWrapp (None, 10)                20295
=================================================================
Total params: 20,448
Trainable params: 20,410
Non-trainable params: 38
_________________________________________________________________
```

You may have noticed a slight difference in the model summary above compared to the baseline model summary in the earlier sections. The total params count increased as expected because of the nodes added by the `quantize_model()` method.

With that, you can now train the model. You will notice that the accuracy is a bit lower because the model is simulating the loss of precision. The training will take a bit longer if you want to achieve the same training accuracy as the earlier run. For this exercise though, we will keep to 1 epoch.

In [20]:
```
# Train the model
q_aware_model.fit(train_images, train_labels, epochs=1, shuffle=False)
```

```
1875/1875 [==============================] - 26s 13ms/step - loss: 0.3028 - accuracy: 0.9158
```

Out[20]:
```
<tensorflow.python.keras.callbacks.History at 0x7eff95fc7650>
```

You can then get the accuracy of the Keras model before and after quantizing the model. The accuracy is expected to be nearly identical because the model is trained to counter the effects of quantization.

In [21]:
```
# Reinitialize the dictionary
ACCURACY = {}

# Get the accuracy of the quantization aware trained model (not yet quantized)
_, ACCURACY['quantization aware non-quantized'] = q_aware_model.evaluate(test_images, test_labels, ve
print_metric(ACCURACY, 'test accuracy')
```

```
test accuracy for quantization aware non-quantized: 0.955299973487854
```

In [22]:
```
# Convert and quantize the model.
convert_tflite(q_aware_model, FILE_QAT_QUANTIZED, quantize=True)

# Get the accuracy of the quantized model
ACCURACY['quantization aware quantized'] = evaluate_tflite_model(FILE_QAT_QUANTIZED, test_images, tes
print_metric(ACCURACY, 'test accuracy')
```

```
WARNING:absl:Found untraced functions such as reshape_1_layer_call_fn, reshape_1_layer_call_and_return
_conditional_losses, conv2d_1_layer_call_fn, conv2d_1_layer_call_and_return_conditional_losses, flatte
n_1_layer_call_fn while saving (showing 5 of 20). These functions will not be directly callable after
loading.
INFO:tensorflow:Assets written to: /tmp/tmpwxa3_k7b/assets
INFO:tensorflow:Assets written to: /tmp/tmpwxa3_k7b/assets
test accuracy for quantization aware non-quantized: 0.955299973487854
test accuracy for quantization aware quantized: 0.9553
```

Pruning

Let's now move on to another technique for reducing model size: Pruning. This process involves zeroing out insignificant (i.e. low magnitude) weights. The intuition is these weights do not contribute as much to making predictions so you can remove them and get the same result. Making the weights sparse helps in compressing the model more efficiently and you will see that in this section.

The Tensorflow Model Optimization Toolkit again has a convenience method for this. The prune_low_magnitude() method puts wrappers in a Keras model so it can be pruned during training. You will pass in the baseline model that you already trained earlier. You will notice that the model summary show increased params because of the wrapper layers added by the pruning method.

You can set how the pruning is done during training. Below, you will use PolynomialDecay to indicate how the sparsity ramps up with each step. Another option available in the library is Constant Sparsity.

In [23]:
```
# Get the pruning method
prune_low_magnitude = tfmot.sparsity.keras.prune_low_magnitude

# Compute end step to finish pruning after 2 epochs.
batch_size = 128
epochs = 2
validation_split = 0.1 # 10% of training set will be used for validation set.

num_images = train_images.shape[0] * (1 - validation_split)
end_step = np.ceil(num_images / batch_size).astype(np.int32) * epochs

# Define pruning schedule.
pruning_params = {
      'pruning_schedule': tfmot.sparsity.keras.PolynomialDecay(initial_sparsity=0.50,
                                                               final_sparsity=0.80,
                                                               begin_step=0,
                                                               end_step=end_step)
}

# Pass in the trained baseline model
model_for_pruning = prune_low_magnitude(baseline_model, **pruning_params)

# `prune_low_magnitude` requires a recompile.
model_for_pruning.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model_for_pruning.summary()
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/base_layer.py:2191: UserWarning:
`layer.add_variable` is deprecated and will be removed in a future version. Please use `layer.add_weig
ht` method instead.
  warnings.warn('`layer.add_variable` is deprecated and '
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
prune_low_magnitude_reshape  (None, 28, 28, 1)         1
_________________________________________________________________
prune_low_magnitude_conv2d ( (None, 26, 26, 12)        230
_________________________________________________________________
prune_low_magnitude_max_pool (None, 13, 13, 12)        1
_________________________________________________________________
prune_low_magnitude_flatten  (None, 2028)              1
_________________________________________________________________
prune_low_magnitude_dense (P (None, 10)                40572
=================================================================
Total params: 40,805
Trainable params: 20,410
Non-trainable params: 20,395
_________________________________________________________________
```

You can also peek at the weights of one of the layers in your model. After pruning, you will notice that many of these will be zeroed out.

In [24]:
```
# Preview model weights
model_for_pruning.weights[1]
```

Out[24]:
```
<tf.Variable 'conv2d/kernel:0' shape=(3, 3, 1, 12) dtype=float32, numpy=
array([[[[ 0.29842487,  0.27442494,  0.4690776 ,  0.23687331,
          -0.7129853 ,  0.3671631 ,  0.40346694, -0.16761062,
          -0.08591487,  0.13260755,  0.04826403,  0.2743174 ]],

        [[ 0.40142423, -0.15539975,  0.5030686 ,  0.1710291 ,
          -0.70189726, -0.05914859,  0.4170224 ,  0.14447062,
           0.24018596,  0.1633075 ,  0.1921253 ,  0.10036336]],

        [[ 0.18591361, -0.34293765,  0.21699017, -0.04197423,
          -0.7483598 , -0.29194227, -0.30614528, -0.07523701,
           0.20286076, -0.14810987, -0.32801974, -0.08863396]]],


       [[[ 0.10306336,  0.11052755, -0.19897152,  0.26521534,
           0.20192622,  0.14787246,  0.62839323,  0.20402573,
           0.00839784,  0.24217863,  0.07323285,  0.04290706]],

        [[-0.03466187,  0.25639522, -0.02021474, -0.04385101,
           0.00818605,  0.03483288,  0.11316509,  0.385911  ,
           0.36726907,  0.08439611, -0.01178847,  0.25763246]],

        [[ 0.01954291, -0.06532311, -0.2202846 ,  0.12646982,
          -0.0977347 ,  0.16638358, -0.74951804,  0.2873893 ,
           0.31843013,  0.03145093, -0.12502334,  0.3158034 ]]],


       [[[-0.4349585 ,  0.25392058, -0.4981905 ,  0.2568549 ,
           0.52235097,  0.10720555,  0.28217515, -0.28670135,
           0.16539723,  0.23085108,  0.29918632, -0.01959905]],

        [[-0.39342687, -0.05594271, -0.51890093,  0.2219396 ,
           0.6102886 ,  0.07331396, -0.47042003, -0.06763031,
           0.16453995, -0.01180941,  0.2273348 , -0.12744649]],

        [[ 0.03587314,  0.10827567, -0.6151041 ,  0.120688  ,
           0.4796519 ,  0.08683834, -0.6724116 ,  0.27692702,
           0.07289029,  0.1906387 ,  0.20458527,  0.23416308]]]],
      dtype=float32)>
```

With that, you can now start re-training the model. Take note that the UpdatePruningStep() callback is required.

In [25]:
```
# Callback to update pruning wrappers at each step
callbacks = [
  tfmot.sparsity.keras.UpdatePruningStep(),
]

# Train and prune the model
model_for_pruning.fit(train_images, train_labels,
                  epochs=epochs, validation_split=validation_split,
                  callbacks=callbacks)
```

```
Epoch 1/2
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/python/ops/array_ops.py:504
9: calling gather (from tensorflow.python.ops.array_ops) with validate_indices is deprecated and will
be removed in a future version.
Instructions for updating:
The `validate_indices` argument has no effect. Indices are always validated on CPU and never validated
on GPU.
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/python/ops/array_ops.py:504
9: calling gather (from tensorflow.python.ops.array_ops) with validate_indices is deprecated and will
be removed in a future version.
Instructions for updating:
The `validate_indices` argument has no effect. Indices are always validated on CPU and never validated
on GPU.
1688/1688 [==============================] - 22s 12ms/step - loss: 0.1710 - accuracy: 0.9539 - val_los
s: 0.1143 - val_accuracy: 0.9700
Epoch 2/2
1688/1688 [==============================] - 20s 12ms/step - loss: 0.1317 - accuracy: 0.9624 - val_los
s: 0.1030 - val_accuracy: 0.9713
```

Out[25]:
```
<tensorflow.python.keras.callbacks.History at 0x7eff9317a090>
```

Now see how the weights in the same layer looks like after pruning.

In [26]:
```
# Preview model weights
model_for_pruning.weights[1]
```

Out[26]:
```
<tf.Variable 'conv2d/kernel:0' shape=(3, 3, 1, 12) dtype=float32, numpy=
array([[[[ 0.        ,  0.        ,  0.7142407 ,  0.        ,
          -0.83520883,  0.78897566,  0.        ,  0.        ,
           0.        ,  0.        ,  0.        ,  0.        ]],
```

```
        [[ 0.73973763,  0.        ,  0.91093683,  0.        ,
          -0.9881885 ,  0.        ,  0.        ,  0.        ,
           0.        ,  0.        ,  0.        ,  0.        ]],

        [[ 0.        ,  0.        ,  0.        ,  0.        ,
          -1.2495996 ,  0.        , -0.        ,  0.        ,
           0.        ,  0.        , -0.        ,  0.        ]]],


       [[[ 0.        ,  0.        ,  0.        ,  0.        ,
           0.        ,  0.        ,  1.3108463 ,  0.        ,
           0.        ,  0.        ,  0.        ,  0.        ]],

        [[-0.        ,  0.        ,  0.        ,  0.        ,
           0.        ,  0.        ,  0.        ,  0.90619195,
           0.7597866 ,  0.        ,  0.        ,  0.6982564 ]],

        [[-0.        ,  0.        ,  0.        ,  0.        ,
          -0.        ,  0.        , -0.9529409 ,  0.7785547 ,
           0.7249059 ,  0.        ,  0.        ,  0.74743104]]],


       [[[-0.        ,  0.        , -1.2707998 ,  0.        ,
           0.9865181 ,  0.        ,  0.        , -0.        ,
           0.        ,  0.        ,  0.        , -0.        ]],

        [[-0.        ,  0.        , -0.9331237 ,  0.        ,
           0.90711576,  0.        , -0.        ,  0.        ,
           0.        ,  0.        ,  0.        ,  0.        ]],

        [[-0.        ,  0.        , -1.2851416 ,  0.        ,
           0.5202148 ,  0.        , -1.3020395 ,  0.        ,
           0.        ,  0.        ,  0.        ,  0.        ]]]],
      dtype=float32)>
```

After pruning, you can remove the wrapper layers to have the same layers and params as the baseline model. You can do that with the strip_pruning() method as shown below. You will do this so you can save the model and also export to TF Lite format just like in the previous sections.

In [27]:
```
# Remove pruning wrappers
model_for_export = tfmot.sparsity.keras.strip_pruning(model_for_pruning)
model_for_export.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
reshape (Reshape)            (None, 28, 28, 1)         0
_________________________________________________________________
conv2d (Conv2D)              (None, 26, 26, 12)        120
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 12)        0
_________________________________________________________________
flatten (Flatten)            (None, 2028)              0
_________________________________________________________________
dense (Dense)                (None, 10)                20290
=================================================================
Total params: 20,410
Trainable params: 20,410
Non-trainable params: 0
_________________________________________________________________
```

You will see the same model weights but the index is different because the wrappers were removed.

In [28]:
```
# Preview model weights (index 1 earlier is now 0 because pruning wrappers were removed)
model_for_export.weights[0]
```

Out[28]:
```
<tf.Variable 'conv2d/kernel:0' shape=(3, 3, 1, 12) dtype=float32, numpy=
array([[[[ 0.        ,  0.        ,  0.7142407 ,  0.        ,
          -0.83520883,  0.78897566,  0.        ,  0.        ,
           0.        ,  0.        ,  0.        ,  0.        ]],

        [[ 0.73973763,  0.        ,  0.91093683,  0.        ,
          -0.9881885 ,  0.        ,  0.        ,  0.        ,
           0.        ,  0.        ,  0.        ,  0.        ]],

        [[ 0.        ,  0.        ,  0.        ,  0.        ,
          -1.2495996 ,  0.        , -0.        ,  0.        ,
           0.        ,  0.        , -0.        ,  0.        ]]],


       [[[ 0.        ,  0.        ,  0.        ,  0.        ,
           0.        ,  0.        ,  1.3108463 ,  0.        ,
           0.        ,  0.        ,  0.        ,  0.        ]],

        [[-0.        ,  0.        ,  0.        ,  0.        ,
```

```
          0.        ,  0.        ,  0.        ,  0.90619195,
          0.7597866 ,  0.        ,  0.        ,  0.6982564 ]],

        [[-0.        ,  0.        ,  0.        ,  0.        ,
          -0.        ,  0.        , -0.9529409 ,  0.7785547 ,
          0.7249059 ,  0.        ,  0.        ,  0.74743104]]],


       [[[-0.        ,  0.        , -1.2707998 ,  0.        ,
          0.9865181 ,  0.        ,  0.        , -0.        ,
          0.        ,  0.        ,  0.        , -0.        ]],

        [[-0.        ,  0.        , -0.9331237 ,  0.        ,
          0.90711576,  0.        , -0.        ,  0.        ,
          0.        ,  0.        ,  0.        ,  0.        ]],

        [[-0.        ,  0.        , -1.2851416 ,  0.        ,
          0.5202148 ,  0.        , -1.3020395 ,  0.        ,
          0.        ,  0.        ,  0.        ,  0.        ]]]],
      dtype=float32)>
```

You will notice below that the pruned model will have the same file size as the baseline_model when saved as H5. This is to be expected. The improvement will be noticeable when you compress the model as will be shown in the cell after this.

In [29]:

```
# Save Keras model
model_for_export.save(FILE_PRUNED_MODEL_H5, include_optimizer=False)

# Get uncompressed model size of baseline and pruned models
MODEL_SIZE = {}
MODEL_SIZE['baseline h5'] = os.path.getsize(FILE_NON_QUANTIZED_H5)
MODEL_SIZE['pruned non quantized h5'] = os.path.getsize(FILE_PRUNED_MODEL_H5)

print_metric(MODEL_SIZE, 'model_size in bytes')
```

```
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.co
mpile_metrics` will be empty until you train or evaluate the model.
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.co
mpile_metrics` will be empty until you train or evaluate the model.
model_size in bytes for baseline h5: 98136
model_size in bytes for pruned non quantized h5: 98136
```

You will use the `get_gzipped_model_size()` helper function in the `Utilities` to compress the models and get its resulting file size. You will notice that the pruned model is about 3 times smaller. This is because of the sparse weights generated by the pruning process. The zeros can be compressed much more efficiently than the low magnitude weights before pruning.

In [30]:

```
# Get compressed size of baseline and pruned models
MODEL_SIZE = {}
MODEL_SIZE['baseline h5'] = get_gzipped_model_size(FILE_NON_QUANTIZED_H5)
MODEL_SIZE['pruned non quantized h5'] = get_gzipped_model_size(FILE_PRUNED_MODEL_H5)

print_metric(MODEL_SIZE, "gzipped model size in bytes")
```

```
gzipped model size in bytes for baseline h5: 77988
gzipped model size in bytes for pruned non quantized h5: 25831
```

You can make the model even more lightweight by quantizing the pruned model. This achieves around 10X reduction in compressed model size as compared to the baseline.

In [31]:

```
# Convert and quantize the pruned model.
pruned_quantized_tflite = convert_tflite(model_for_export, FILE_PRUNED_QUANTIZED_TFLITE, quantize=True

# Compress and get the model size
MODEL_SIZE['pruned quantized tflite'] = get_gzipped_model_size(FILE_PRUNED_QUANTIZED_TFLITE)
print_metric(MODEL_SIZE, "gzipped model size in bytes")
```

```
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.co
mpile_metrics` will be empty until you train or evaluate the model.
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.co
mpile_metrics` will be empty until you train or evaluate the model.
INFO:tensorflow:Assets written to: /tmp/tmp689cw60i/assets
INFO:tensorflow:Assets written to: /tmp/tmp689cw60i/assets
gzipped model size in bytes for baseline h5: 77988
gzipped model size in bytes for pruned non quantized h5: 25831
gzipped model size in bytes for pruned quantized tflite: 8306
```

As expected, the TF Lite model's accuracy will also be close to the Keras model.

In [32]:

```
# Get accuracy of pruned Keras and TF Lite models
ACCURACY = {}

_, ACCURACY['pruned model h5'] = model_for_pruning.evaluate(test_images, test_labels)
ACCURACY['pruned and quantized tflite'] = evaluate_tflite_model(FILE_PRUNED_QUANTIZED_TFLITE, test_ima

print_metric(ACCURACY, 'accuracy')
```

```
313/313 [==============================] - 2s 6ms/step - loss: 0.1222 - accuracy: 0.9624
accuracy for pruned model h5: 0.9624000191688538
accuracy for pruned and quantized tflite: 0.9626
```

Wrap Up

In this notebook, you practiced several techniques in optimizing your models for mobile and embedded applications. You used quantization to reduce floating point representations into integer, then used pruning to make the weights sparse for efficient model compression. These make your models lightweight for efficient transport and storage without sacrificing model accuracy. Try this in your own models and see what performance you get. For more information, here are a few other resources:

- Post Training Quantization Guide
- Quantization Aware Training Comprehensive Guide
- Pruning Comprehensive Guide

Congratulations and enjoy the rest of the course!