Image segmentation and neural network quantization

Here are all the import statements needed for all the exercises

```
%tensorflow_version 2.x
import tensorflow as tf
%load_ext tensorboard
import numpy as np
import matplotlib.pyplot as plt

! pip install tensorflow_model_optimization
import tensorflow_model_optimization as tfmot
```

```
Colab only includes TensorFlow 2.x; %tensorflow_version has no effect.
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
Requirement already satisfied: tensorflow_model_optimization in /usr/local/lib/python3.10/dist-packages (0.8.0)
Requirement already satisfied: absl-py~=1.2 in /usr/local/lib/python3.10/dist-packages (from tensorflow_model_optimization) (1.4.0)
Requirement already satisfied: dm-tree~=0.1.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow_model_optimization) (0.1.8)
Requirement already satisfied: numpy~=1.23 in /usr/local/lib/python3.10/dist-packages (from tensorflow_model_optimization) (1.26.4)
Requirement already satisfied: six~=1.14 in /usr/local/lib/python3.10/dist-packages (from tensorflow_model_optimization) (1.17.0)
```

Loading the Sentinel 2 dataset. Images and labels are padded to be 256x256 in size and normalized by their maximum value. 40 images are used for the train partition (X_train, Y_train) and 10 for testing (X_test, Y_test)

```
loaded = np.load('sentinel2.npz')
X = loaded['X'].astype(np.float32)  # images dataset
Y = loaded['Y'].astype(np.float32)  # class dataset associated to each pixel (1 means coltivated land, 0 otherwise)
X = np.pad(X, ((0,0),(3,3),(3,3),(0,0)))
Y = np.pad(Y, ((0,0),(3,3),(3,3),(0,0)))
X_train = X[:40]/np.max(X[:40]) # divedes datasets into training and testing
X_test = X[40:]/np.max(X[:40])
Y_train = Y[:40]
Y_test = Y[40:]
```

```
# from google.colab import drive
# drive.mount('/content/drive')
```

[TODO] Implement the U-net neural network for segmentation as drawn in the lab document.

```
# The structure of the U-net NN follows the one that we have seen in the lecture. In the lecture we have seen the Res-Net NN that introduces skip connections between layers.
# This structure is useful cause the NN can better understand the relations between inputs and outputs, and it can perform channel attention or spatial attention.
# Channel attention consists in weighting the feature map with a coefficent evaluated on the input of the layer.
# Spatial attention is very similar and consists in weighting the pixels shared on the features of the image.
# When we introduce skip connection we need to concatenate two output togheter on the features space, for doing so we must consider that they should have the same shape HxW.
# In this NN we see also how encoding and decoding works. Encoding is used to perform features extraction through convolutional layers that reduce the size of the image but i
# of feature maps. In this case for reducing the size of the image we are changing the strides. Strides means the jump that the kernel does on the image during the convolutio
# value equals to 2 means that we are halving the size of the input image.
# Decondig is used to recontruct the image from the features extracted through the encoding. The deconding operation is used also for images segmentation.
# In this case the deconding operation recontruct the image through concatenation with some encoding outputs for adding new features and so more details and then it uses UpSa
# for increasing the size of the image. This is done by adding zeros around the image and then interpolate them with the other pixels.

def unet(input_shape):
  # first layer
  inputs = tf.keras.Input(shape=input_shape)
  x = tf.keras.layers.Conv2D(64, 3, strides=1, activation='relu', padding='same')(inputs) # more we go deeper on the levels and more we increase the number of feature maps e
  x = tf.keras.layers.BatchNormalization()(x) # batch normalization performs a standardization of the input of the layer by using statistics evaluated on a batch. This is use
  x = tf.keras.layers.ReLU()(x) # non linear activation function, important cause a linear activation function would make the NN a linear model and so we would lost the adva
  # the output of this layer is H x W x 64
  # second layer - concatenated to the first
  x = tf.keras.layers.Conv2D(64, 3, strides=1, activation='relu', padding='same')(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.ReLU()(x)
  # the output of this layer is H x W x 64
  # third layer
  x1 = tf.keras.layers.Conv2D(128, 3, strides=2, activation='relu', padding='same')(x)  # strides = 2 means that the kernel of the convolution will jump 2 pixels at a time o
  x1 = tf.keras.layers.BatchNormalization()(x1)
  x1 = tf.keras.layers.ReLU()(x1)
  # the output of this layer is H/2 x W/2 x 128
  # fourth layer
  x2 = tf.keras.layers.Conv2D(256, 3, strides=2, activation='relu', padding='same')(x1)
  x2 = tf.keras.layers.BatchNormalization()(x2)
  x2 = tf.keras.layers.ReLU()(x2)
  # the output of this layer is H/4 x W/4 x 256
  # fifth layer - concatenated to the fourth
  x2 = tf.keras.layers.Conv2D(256, 3, strides=1, activation='relu', padding='same')(x2)
  x2 = tf.keras.layers.BatchNormalization()(x2)
  x2 = tf.keras.layers.ReLU()(x2)
  x2 = tf.keras.layers.UpSampling2D(size=(2,2))(x2) # upsample the image by doubling both the dimensions
  x2 = tf.keras.layers.concatenate([x2, x1])  # concatenate the output of the layer with the output of the third layer by the features space, they should have the same image
  # the output of this layer is H/2 x W/2 x 256+128
  # sixth layer
  x3 = tf.keras.layers.Conv2D(128, 3, strides=1, activation='relu', padding='same')(x2)
  x3 = tf.keras.layers.BatchNormalization()(x3)
  x3 = tf.keras.layers.ReLU()(x3)
  x3 = tf.keras.layers.UpSampling2D(size=(2,2))(x3)
  x3 = tf.keras.layers.concatenate([x3, x]) # concatenate the output of the layer with the output of the first-second layer by the features space, they should have the same i
  # the output of this layer is H x W x 128+64
  # seventh layer
  x4 = tf.keras.layers.Conv2D(64, 3, strides=1, activation='relu', padding='same')(x3)
  x4 = tf.keras.layers.BatchNormalization()(x4)
  x4 = tf.keras.layers.ReLU()(x4)
  # the output of this layer is H x W x 64

  # note that the last layer is not a Dense layer that performs the classification. This because we are performing image segmentation and not image classification, so the re
  outputs = tf.keras.layers.Conv2D(2, 1, activation='softmax')(x4)  # softmax will create a pdf between the two possible classes, dog or cat
  #outputs = tf.keras.layers.Conv2D(1, 1, activation='sigmoid')(x4) # sigmoid maps the input value in a probability between 0 and 1, it doesn't build a pdf. It return the cl
  model = tf.keras.Model(inputs=inputs, outputs=outputs)
  return model

Unet_model = unet((256,256,12)) # the input shape is 256x256x12
Unet_model.summary()
```

```
 chNormalization)

 re_lu_2 (ReLU)              (None, 128, 128, 128)        0         ['batch_normalization_2[0][0]'
                                                                     ]

 conv2d_3 (Conv2D)           (None, 64, 64, 256)          295168    ['re_lu_2[0][0]']

 batch_normalization_3 (Bat  (None, 64, 64, 256)          1024      ['conv2d_3[0][0]']
 chNormalization)

 re_lu_3 (ReLU)              (None, 64, 64, 256)          0         ['batch_normalization_3[0][0]'
                                                                     ]

 conv2d_4 (Conv2D)           (None, 64, 64, 256)          590080    ['re_lu_3[0][0]']

 batch_normalization_4 (Bat  (None, 64, 64, 256)          1024      ['conv2d_4[0][0]']
 chNormalization)

 re_lu_4 (ReLU)              (None, 64, 64, 256)          0         ['batch_normalization_4[0][0]'
                                                                     ]

 up_sampling2d (UpSampling2  (None, 128, 128, 256)        0         ['re_lu_4[0][0]']
 D)

 concatenate (Concatenate)   (None, 128, 128, 384)        0         ['up_sampling2d[0][0]',
                                                                      're_lu_2[0][0]']

 conv2d_5 (Conv2D)           (None, 128, 128, 128)        442496    ['concatenate[0][0]']

 batch_normalization_5 (Bat  (None, 128, 128, 128)        512       ['conv2d_5[0][0]']
 chNormalization)

 re_lu_5 (ReLU)              (None, 128, 128, 128)        0         ['batch_normalization_5[0][0]'
                                                                     ]

 up_sampling2d_1 (UpSamplin  (None, 256, 256, 128)        0         ['re_lu_5[0][0]']
 g2D)

 concatenate_1 (Concatenate  (None, 256, 256, 192)        0         ['up_sampling2d_1[0][0]',
 )                                                                    're_lu_1[0][0]']

 conv2d_6 (Conv2D)           (None, 256, 256, 64)         110656    ['concatenate_1[0][0]']

 batch_normalization_6 (Bat  (None, 256, 256, 64)         256       ['conv2d_6[0][0]']
 chNormalization)

 re_lu_6 (ReLU)              (None, 256, 256, 64)         0         ['batch_normalization_6[0][0]'
                                                                     ]

 conv2d_7 (Conv2D)           (None, 256, 256, 2)          130       ['re_lu_6[0][0]']

==================================================================================================
Total params: 1560130 (5.95 MB)
Trainable params: 1558210 (5.94 MB)
Non-trainable params: 1920 (7.50 KB)
```

[TODO] Compile and train the model (might take some time...)

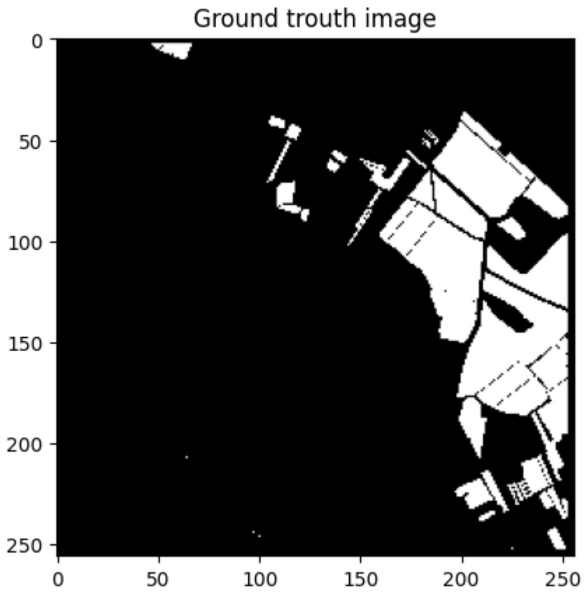
```
Unet_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.01),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
Unet_model.fit(X_train, Y_train, epochs=50)
```

```
2/2 [==============================] - 1s 285ms/step - loss: 0.3021 - accuracy: 0.8572
Epoch 23/50
2/2 [==============================] - 1s 286ms/step - loss: 0.2965 - accuracy: 0.8619
Epoch 24/50
2/2 [==============================] - 2s 291ms/step - loss: 0.2858 - accuracy: 0.8675
Epoch 25/50
2/2 [==============================] - 1s 288ms/step - loss: 0.2851 - accuracy: 0.8707
Epoch 26/50
2/2 [==============================] - 1s 281ms/step - loss: 0.2759 - accuracy: 0.8729
Epoch 27/50
2/2 [==============================] - 1s 296ms/step - loss: 0.2761 - accuracy: 0.8767
Epoch 28/50
2/2 [==============================] - 1s 298ms/step - loss: 0.2775 - accuracy: 0.8740
Epoch 29/50
2/2 [==============================] - 1s 300ms/step - loss: 0.2790 - accuracy: 0.8767
Epoch 30/50
2/2 [==============================] - 1s 286ms/step - loss: 0.2713 - accuracy: 0.8805
Epoch 31/50
2/2 [==============================] - 1s 287ms/step - loss: 0.2631 - accuracy: 0.8829
Epoch 32/50
2/2 [==============================] - 1s 288ms/step - loss: 0.2585 - accuracy: 0.8832
Epoch 33/50
2/2 [==============================] - 1s 292ms/step - loss: 0.2556 - accuracy: 0.8890
Epoch 34/50
2/2 [==============================] - 1s 309ms/step - loss: 0.2536 - accuracy: 0.8902
Epoch 35/50
2/2 [==============================] - 1s 289ms/step - loss: 0.2396 - accuracy: 0.8939
Epoch 36/50
2/2 [==============================] - 1s 293ms/step - loss: 0.2471 - accuracy: 0.8898
Epoch 37/50
2/2 [==============================] - 1s 301ms/step - loss: 0.2364 - accuracy: 0.8965
Epoch 38/50
2/2 [==============================] - 1s 296ms/step - loss: 0.2376 - accuracy: 0.8979
Epoch 39/50
2/2 [==============================] - 1s 284ms/step - loss: 0.2432 - accuracy: 0.8978
Epoch 40/50
2/2 [==============================] - 1s 288ms/step - loss: 0.2369 - accuracy: 0.8984
Epoch 41/50
2/2 [==============================] - 1s 290ms/step - loss: 0.2376 - accuracy: 0.8975
Epoch 42/50
2/2 [==============================] - 1s 294ms/step - loss: 0.2278 - accuracy: 0.9018
Epoch 43/50
2/2 [==============================] - 1s 292ms/step - loss: 0.2336 - accuracy: 0.8996
Epoch 44/50
2/2 [==============================] - 1s 289ms/step - loss: 0.2289 - accuracy: 0.9019
Epoch 45/50
2/2 [==============================] - 1s 291ms/step - loss: 0.2261 - accuracy: 0.9038
Epoch 46/50
2/2 [==============================] - 1s 286ms/step - loss: 0.2231 - accuracy: 0.9051
Epoch 47/50
2/2 [==============================] - 1s 285ms/step - loss: 0.2231 - accuracy: 0.9059
Epoch 48/50
2/2 [==============================] - 1s 286ms/step - loss: 0.2222 - accuracy: 0.9059
Epoch 49/50
2/2 [==============================] - 1s 281ms/step - loss: 0.2282 - accuracy: 0.9053
Epoch 50/50
2/2 [==============================] - 1s 282ms/step - loss: 0.2232 - accuracy: 0.9040
<tf_keras.src.callbacks.History at 0x7bfe8443fb20>
```

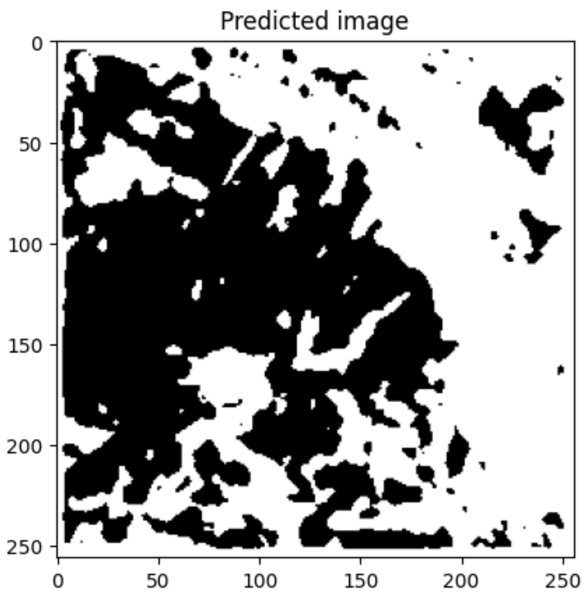
[TODO] Test the model on the test set and measure the accuracy.

```
Unet_model.evaluate(X_test, Y_test)

image = Y_test[1]
plt.imshow(image, 'gray')
plt.title('Ground trouth image')
plt.show()

predict_im = Unet_model.predict(X_test[1].reshape(1,256,256,12))
predict_im = np.squeeze(predict_im)
predict_im = np.argmax(predict_im, axis=-1)
plt.imshow(predict_im, 'gray')
plt.title('Predicted image')
plt.show()
```

```
1/1 [==============================] - 2s 2s/step - loss: 0.9591 - accuracy: 0.5645
```




```
1/1 [==============================] - 1s 599ms/step
```



[TO DO] Convert model to TFLite with 8-bit weight quantization

```
# in our model we have many parameters and each is saved as float32. This cost to use a lot of memory space, so we want to reduce the dimension of these parameters. For doin(
# perform quantization, so we can quantize each float parameter to an int parameter that can be rappresented with only 8 bits.
# PTQ (Post Training Quantization) quantize each parameter on an already trained neural network. This type of quantization is the faster and the easiest to implement, but it
# accuracy of the model, cause the quantization is chaning the optimal parameters that the model has found during the training.

converter = tf.lite.TFLiteConverter.from_keras_model(Unet_model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]

quantized_model = converter.convert()
with open("quantized_model.tflite", "wb") as f:
  f.write(quantized_model)
```

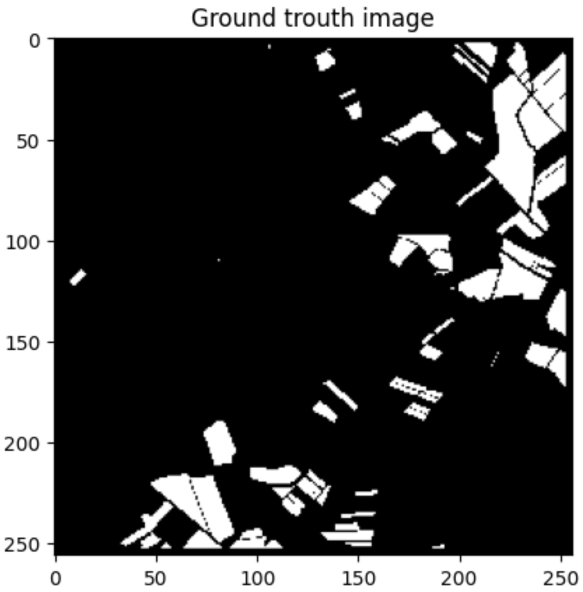
[TODO] Test the accuracy of the quantized model by writing your own "evaluate" function. Remember that TFLite interpreter can only process one sample at a time, not a batch.

```
# Initialize the interpreter
interpreter = tf.lite.Interpreter('quantized_model.tflite')
interpreter.allocate_tensors()

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

def evaluate(interpreter, X_test, Y_test):
  accuracies = []

  for i, (input_data, true_class) in enumerate(zip(X_test, Y_test)):
    input_data = input_data.astype(input_details[0]['dtype']) # read the image from the test dataset

    interpreter.set_tensor(input_details[0]['index'], np.expand_dims(input_data, axis=0)) # set the input tensor with the image and add a dimension for the batch

    interpreter.invoke()  # run the inference on the image
    output_data = interpreter.get_tensor(output_details[0]['index'])  # get the output tensor of the model
    predicted_class = np.squeeze(output_data) # remove the batch dimension
    predicted_class = np.argmax(predicted_class, axis=2)  # get the class with the highest probability

    true_class = np.squeeze(true_class) # remove the batch dimension
    accuracy = np.mean(predicted_class == true_class) # evaluate the accuracy of the model
    accuracies.append(accuracy) # save the accuracy of the model

    print(f"Image {i+1} - Accuracy: {accuracy}")  # print the accuracy of the model

    plt.imshow(true_class, 'gray')  # show the image
```

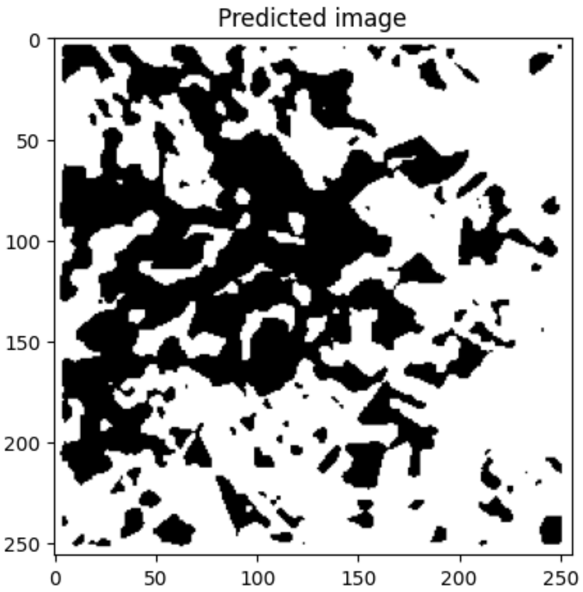
```
        plt.imshow(true_class, 'gray') # show the image
        plt.title('Ground trouth image')
        plt.show()

        plt.imshow(predicted_class, 'gray') # show the predicted image
        plt.title('Predicted image')
        plt.show()
      return np.mean(accuracies)
    # Call the evaluate function
    accuracy = evaluate(interpreter, X_test, Y_test)
    print(f"Mean accuracy: {accuracy}")

    original_predictions = Unet_model.predict(X_test)
    original_predictions = (original_predictions > 0.5).astype(np.uint8)
```

```
Image 1 - Accuracy: 0.4641876220703125
```

Ground trouth image

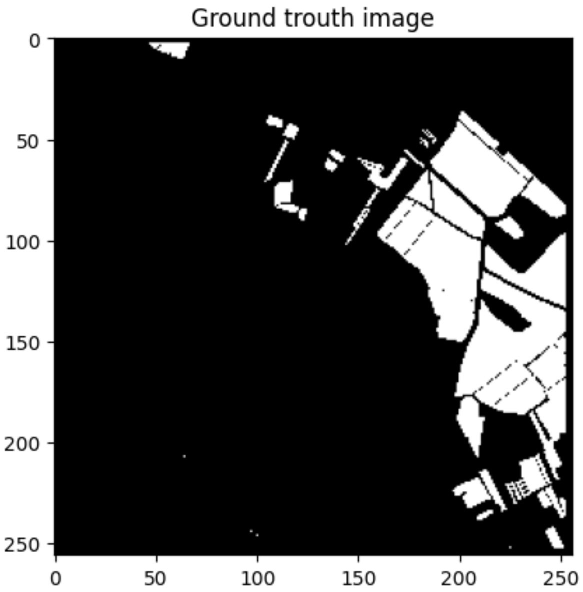


Predicted image

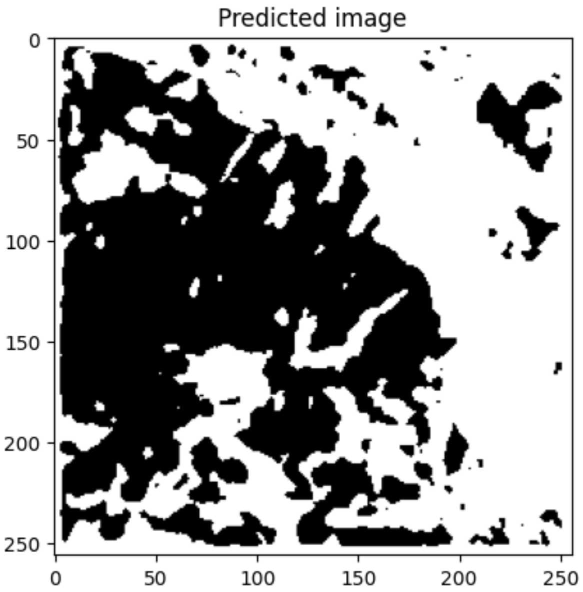


```
Image 2 - Accuracy: 0.5809478759765625
```

Ground trouth image



Predicted image



```
Image 3 - Accuracy: 0.6994171142578125
```

Ground trouth image