lab3

January 21, 2025

1 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/python3.11/dist-packages (0.8.0)
Requirement already satisfied: absl-py~=1.2 in /usr/local/lib/python3.11/dist-
packages (from tensorflow_model_optimization) (1.4.0)
Requirement already satisfied: dm-tree~=0.1.1 in /usr/local/lib/python3.11/dist-
packages (from tensorflow_model_optimization) (0.1.8)
Requirement already satisfied: numpy~=1.23 in /usr/local/lib/python3.11/dist-
packages (from tensorflow_model_optimization) (1.26.4)
Requirement already satisfied: six~=1.14 in /usr/local/lib/python3.11/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 coeffcent
↪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 increase the number
# 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 convolution, and use this
# 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 UpSampling2D
# 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 extracted, because we want to extract
↪more complex features
  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 useful to avoid the vanishing and expoding gradient problem
```

```python
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 advantages of having deep layers
# 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 on the image, so the result of the convolution will be halved
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 size
# 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 image size
# 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 results of the NN is an image and not a class.
    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 class with the highest probability
    model = tf.keras.Model(inputs=inputs, outputs=outputs)
    return model

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

[**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=80, validation_split=0.2)
```

```
Epoch 1/80
1/1 [==============================] - 4s 4s/step - loss: 0.9394 - accuracy:
0.4735 - val_loss: 4.3618 - val_accuracy: 0.7734
Epoch 2/80
1/1 [==============================] - 1s 1s/step - loss: 0.8931 - accuracy:
0.5527 - val_loss: 86.7708 - val_accuracy: 0.7734
Epoch 3/80
1/1 [==============================] - 1s 1s/step - loss: 0.6388 - accuracy:
0.8248 - val_loss: 354.2708 - val_accuracy: 0.7734
Epoch 4/80
1/1 [==============================] - 1s 1s/step - loss: 0.5232 - accuracy:
0.8317 - val_loss: 594.3196 - val_accuracy: 0.7734
Epoch 5/80
1/1 [==============================] - 1s 1s/step - loss: 0.5105 - accuracy:
0.8309 - val_loss: 682.0712 - val_accuracy: 0.7734
Epoch 6/80
1/1 [==============================] - 1s 1s/step - loss: 0.4629 - accuracy:
0.8399 - val_loss: 710.4044 - val_accuracy: 0.7734
Epoch 7/80
1/1 [==============================] - 1s 1s/step - loss: 0.4232 - accuracy:
0.8433 - val_loss: 682.3586 - val_accuracy: 0.7734
```

```
Epoch 8/80
1/1 [==============================] - 1s 1s/step - loss: 0.3969 - accuracy:
0.8455 - val_loss: 619.4299 - val_accuracy: 0.7734
Epoch 9/80
1/1 [==============================] - 1s 1s/step - loss: 0.3834 - accuracy:
0.8465 - val_loss: 543.3018 - val_accuracy: 0.7734
Epoch 10/80
1/1 [==============================] - 1s 1s/step - loss: 0.3662 - accuracy:
0.8486 - val_loss: 472.1897 - val_accuracy: 0.7734
Epoch 11/80
1/1 [==============================] - 1s 1s/step - loss: 0.3513 - accuracy:
0.8502 - val_loss: 408.8618 - val_accuracy: 0.7734
Epoch 12/80
1/1 [==============================] - 1s 1s/step - loss: 0.3414 - accuracy:
0.8512 - val_loss: 346.7076 - val_accuracy: 0.7734
Epoch 13/80
1/1 [==============================] - 1s 1s/step - loss: 0.3334 - accuracy:
0.8523 - val_loss: 291.6997 - val_accuracy: 0.7734
Epoch 14/80
1/1 [==============================] - 1s 1s/step - loss: 0.3259 - accuracy:
0.8549 - val_loss: 242.5887 - val_accuracy: 0.7734
Epoch 15/80
1/1 [==============================] - 1s 1s/step - loss: 0.3189 - accuracy:
0.8567 - val_loss: 201.5874 - val_accuracy: 0.7734
Epoch 16/80
1/1 [==============================] - 1s 1s/step - loss: 0.3126 - accuracy:
0.8582 - val_loss: 168.2029 - val_accuracy: 0.7734
Epoch 17/80
1/1 [==============================] - 1s 1s/step - loss: 0.3062 - accuracy:
0.8596 - val_loss: 140.6690 - val_accuracy: 0.7734
Epoch 18/80
1/1 [==============================] - 1s 1s/step - loss: 0.3005 - accuracy:
0.8606 - val_loss: 117.5165 - val_accuracy: 0.7734
Epoch 19/80
1/1 [==============================] - 1s 1s/step - loss: 0.2950 - accuracy:
0.8606 - val_loss: 98.0369 - val_accuracy: 0.7734
Epoch 20/80
1/1 [==============================] - 1s 1s/step - loss: 0.2883 - accuracy:
0.8625 - val_loss: 81.8811 - val_accuracy: 0.7734
Epoch 21/80
1/1 [==============================] - 1s 1s/step - loss: 0.2830 - accuracy:
0.8644 - val_loss: 68.4908 - val_accuracy: 0.7734
Epoch 22/80
1/1 [==============================] - 1s 1s/step - loss: 0.2771 - accuracy:
0.8656 - val_loss: 57.3428 - val_accuracy: 0.7734
Epoch 23/80
1/1 [==============================] - 1s 1s/step - loss: 0.2728 - accuracy:
0.8667 - val_loss: 48.1528 - val_accuracy: 0.7734
```

```
Epoch 24/80
1/1 [==============================] - 1s 1s/step - loss: 0.2681 - accuracy:
0.8680 - val_loss: 40.5435 - val_accuracy: 0.7734
Epoch 25/80
1/1 [==============================] - 1s 1s/step - loss: 0.2633 - accuracy:
0.8694 - val_loss: 34.2984 - val_accuracy: 0.7734
Epoch 26/80
1/1 [==============================] - 1s 1s/step - loss: 0.2589 - accuracy:
0.8707 - val_loss: 29.2521 - val_accuracy: 0.7734
Epoch 27/80
1/1 [==============================] - 1s 1s/step - loss: 0.2541 - accuracy:
0.8727 - val_loss: 25.1450 - val_accuracy: 0.7734
Epoch 28/80
1/1 [==============================] - 1s 1s/step - loss: 0.2501 - accuracy:
0.8746 - val_loss: 21.5218 - val_accuracy: 0.7734
Epoch 29/80
1/1 [==============================] - 1s 1s/step - loss: 0.2458 - accuracy:
0.8768 - val_loss: 18.3530 - val_accuracy: 0.7734
Epoch 30/80
1/1 [==============================] - 1s 1s/step - loss: 0.2425 - accuracy:
0.8795 - val_loss: 15.7451 - val_accuracy: 0.7734
Epoch 31/80
1/1 [==============================] - 1s 1s/step - loss: 0.2389 - accuracy:
0.8823 - val_loss: 13.5207 - val_accuracy: 0.7734
Epoch 32/80
1/1 [==============================] - 1s 1s/step - loss: 0.2358 - accuracy:
0.8844 - val_loss: 11.5905 - val_accuracy: 0.7734
Epoch 33/80
1/1 [==============================] - 1s 1s/step - loss: 0.2327 - accuracy:
0.8860 - val_loss: 10.0424 - val_accuracy: 0.7734
Epoch 34/80
1/1 [==============================] - 1s 1s/step - loss: 0.2304 - accuracy:
0.8873 - val_loss: 8.7342 - val_accuracy: 0.7734
Epoch 35/80
1/1 [==============================] - 1s 1s/step - loss: 0.2280 - accuracy:
0.8888 - val_loss: 7.6591 - val_accuracy: 0.7734
Epoch 36/80
1/1 [==============================] - 1s 1s/step - loss: 0.2261 - accuracy:
0.8899 - val_loss: 6.7607 - val_accuracy: 0.7734
Epoch 37/80
1/1 [==============================] - 1s 1s/step - loss: 0.2238 - accuracy:
0.8912 - val_loss: 5.9652 - val_accuracy: 0.7734
Epoch 38/80
1/1 [==============================] - 1s 1s/step - loss: 0.2221 - accuracy:
0.8925 - val_loss: 5.2274 - val_accuracy: 0.7734
Epoch 39/80
1/1 [==============================] - 1s 1s/step - loss: 0.2202 - accuracy:
0.8936 - val_loss: 4.5857 - val_accuracy: 0.7734
```

```
Epoch 40/80
1/1 [==============================] - 1s 1s/step - loss: 0.2184 - accuracy:
0.8949 - val_loss: 4.0290 - val_accuracy: 0.7734
Epoch 41/80
1/1 [==============================] - 1s 1s/step - loss: 0.2166 - accuracy:
0.8959 - val_loss: 3.5116 - val_accuracy: 0.7734
Epoch 42/80
1/1 [==============================] - 1s 1s/step - loss: 0.2148 - accuracy:
0.8972 - val_loss: 3.0580 - val_accuracy: 0.7734
Epoch 43/80
1/1 [==============================] - 1s 1s/step - loss: 0.2133 - accuracy:
0.8980 - val_loss: 2.6641 - val_accuracy: 0.7734
Epoch 44/80
1/1 [==============================] - 1s 1s/step - loss: 0.2115 - accuracy:
0.8996 - val_loss: 2.3222 - val_accuracy: 0.7734
Epoch 45/80
1/1 [==============================] - 1s 1s/step - loss: 0.2099 - accuracy:
0.9005 - val_loss: 2.0152 - val_accuracy: 0.7734
Epoch 46/80
1/1 [==============================] - 1s 1s/step - loss: 0.2076 - accuracy:
0.9028 - val_loss: 1.7560 - val_accuracy: 0.7733
Epoch 47/80
1/1 [==============================] - 1s 1s/step - loss: 0.2056 - accuracy:
0.9039 - val_loss: 1.5425 - val_accuracy: 0.7733
Epoch 48/80
1/1 [==============================] - 1s 1s/step - loss: 0.2042 - accuracy:
0.9049 - val_loss: 1.3449 - val_accuracy: 0.7737
Epoch 49/80
1/1 [==============================] - 1s 1s/step - loss: 0.2027 - accuracy:
0.9068 - val_loss: 1.1777 - val_accuracy: 0.7741
Epoch 50/80
1/1 [==============================] - 1s 1s/step - loss: 0.2012 - accuracy:
0.9074 - val_loss: 1.0287 - val_accuracy: 0.7723
Epoch 51/80
1/1 [==============================] - 1s 1s/step - loss: 0.1995 - accuracy:
0.9094 - val_loss: 0.9269 - val_accuracy: 0.7075
Epoch 52/80
1/1 [==============================] - 1s 1s/step - loss: 0.1981 - accuracy:
0.9097 - val_loss: 0.8369 - val_accuracy: 0.7045
Epoch 53/80
1/1 [==============================] - 1s 1s/step - loss: 0.1963 - accuracy:
0.9116 - val_loss: 0.7847 - val_accuracy: 0.7019
Epoch 54/80
1/1 [==============================] - 1s 1s/step - loss: 0.1947 - accuracy:
0.9131 - val_loss: 0.7569 - val_accuracy: 0.6973
Epoch 55/80
1/1 [==============================] - 1s 1s/step - loss: 0.1936 - accuracy:
0.9133 - val_loss: 0.7182 - val_accuracy: 0.7005
```

```
Epoch 56/80
1/1 [==============================] - 1s 1s/step - loss: 0.1955 - accuracy:
0.9149 - val_loss: 0.7666 - val_accuracy: 0.6932
Epoch 57/80
1/1 [==============================] - 1s 1s/step - loss: 0.2004 - accuracy:
0.9102 - val_loss: 0.7328 - val_accuracy: 0.6995
Epoch 58/80
1/1 [==============================] - 1s 1s/step - loss: 0.1967 - accuracy:
0.9148 - val_loss: 0.7567 - val_accuracy: 0.6950
Epoch 59/80
1/1 [==============================] - 1s 1s/step - loss: 0.1940 - accuracy:
0.9149 - val_loss: 0.7639 - val_accuracy: 0.6923
Epoch 60/80
1/1 [==============================] - 1s 1s/step - loss: 0.1907 - accuracy:
0.9158 - val_loss: 0.7615 - val_accuracy: 0.6947
Epoch 61/80
1/1 [==============================] - 1s 1s/step - loss: 0.1934 - accuracy:
0.9155 - val_loss: 0.7858 - val_accuracy: 0.6914
Epoch 62/80
1/1 [==============================] - 1s 1s/step - loss: 0.1879 - accuracy:
0.9184 - val_loss: 0.8198 - val_accuracy: 0.6887
Epoch 63/80
1/1 [==============================] - 1s 1s/step - loss: 0.1896 - accuracy:
0.9177 - val_loss: 0.8134 - val_accuracy: 0.6918
Epoch 64/80
1/1 [==============================] - 1s 1s/step - loss: 0.1857 - accuracy:
0.9204 - val_loss: 0.8158 - val_accuracy: 0.6946
Epoch 65/80
1/1 [==============================] - 1s 1s/step - loss: 0.1866 - accuracy:
0.9196 - val_loss: 0.8679 - val_accuracy: 0.6834
Epoch 66/80
1/1 [==============================] - 1s 1s/step - loss: 0.1837 - accuracy:
0.9213 - val_loss: 0.9118 - val_accuracy: 0.6751
Epoch 67/80
1/1 [==============================] - 1s 1s/step - loss: 0.1838 - accuracy:
0.9214 - val_loss: 0.8709 - val_accuracy: 0.6875
Epoch 68/80
1/1 [==============================] - 1s 1s/step - loss: 0.1817 - accuracy:
0.9225 - val_loss: 0.8943 - val_accuracy: 0.6803
Epoch 69/80
1/1 [==============================] - 1s 1s/step - loss: 0.1830 - accuracy:
0.9225 - val_loss: 0.8690 - val_accuracy: 0.6860
Epoch 70/80
1/1 [==============================] - 1s 1s/step - loss: 0.1831 - accuracy:
0.9221 - val_loss: 0.9877 - val_accuracy: 0.6543
Epoch 71/80
1/1 [==============================] - 1s 1s/step - loss: 0.1797 - accuracy:
0.9239 - val_loss: 0.9615 - val_accuracy: 0.6565
```

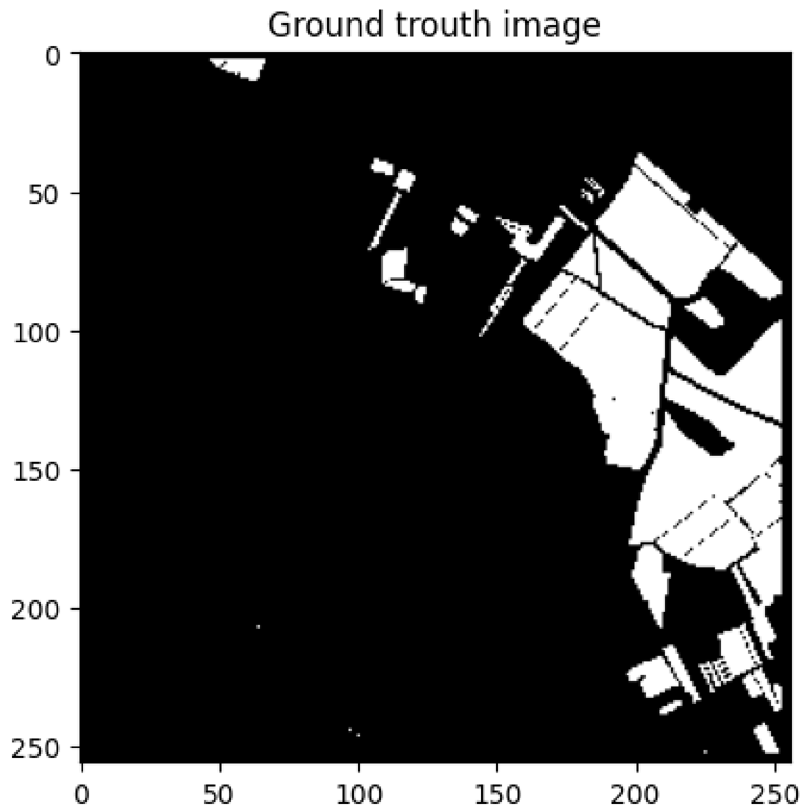
```
Epoch 72/80
1/1 [==============================] - 1s 1s/step - loss: 0.1789 - accuracy:
0.9247 - val_loss: 0.8637 - val_accuracy: 0.6862
Epoch 73/80
1/1 [==============================] - 1s 1s/step - loss: 0.1813 - accuracy:
0.9227 - val_loss: 0.9479 - val_accuracy: 0.6570
Epoch 74/80
1/1 [==============================] - 1s 1s/step - loss: 0.1764 - accuracy:
0.9257 - val_loss: 1.0127 - val_accuracy: 0.6395
Epoch 75/80
1/1 [==============================] - 1s 1s/step - loss: 0.1765 - accuracy:
0.9258 - val_loss: 0.9234 - val_accuracy: 0.6705
Epoch 76/80
1/1 [==============================] - 1s 1s/step - loss: 0.1769 - accuracy:
0.9249 - val_loss: 0.9279 - val_accuracy: 0.6712
Epoch 77/80
1/1 [==============================] - 1s 1s/step - loss: 0.1737 - accuracy:
0.9269 - val_loss: 0.9712 - val_accuracy: 0.6481
Epoch 78/80
1/1 [==============================] - 1s 1s/step - loss: 0.1746 - accuracy:
0.9266 - val_loss: 0.9611 - val_accuracy: 0.6625
Epoch 79/80
1/1 [==============================] - 1s 1s/step - loss: 0.1751 - accuracy:
0.9260 - val_loss: 1.0403 - val_accuracy: 0.6293
Epoch 80/80
1/1 [==============================] - 1s 1s/step - loss: 0.1718 - accuracy:
0.9278 - val_loss: 0.9983 - val_accuracy: 0.6449
```

[ ]: <tf_keras.src.callbacks.History at 0x7ebf8092f790>

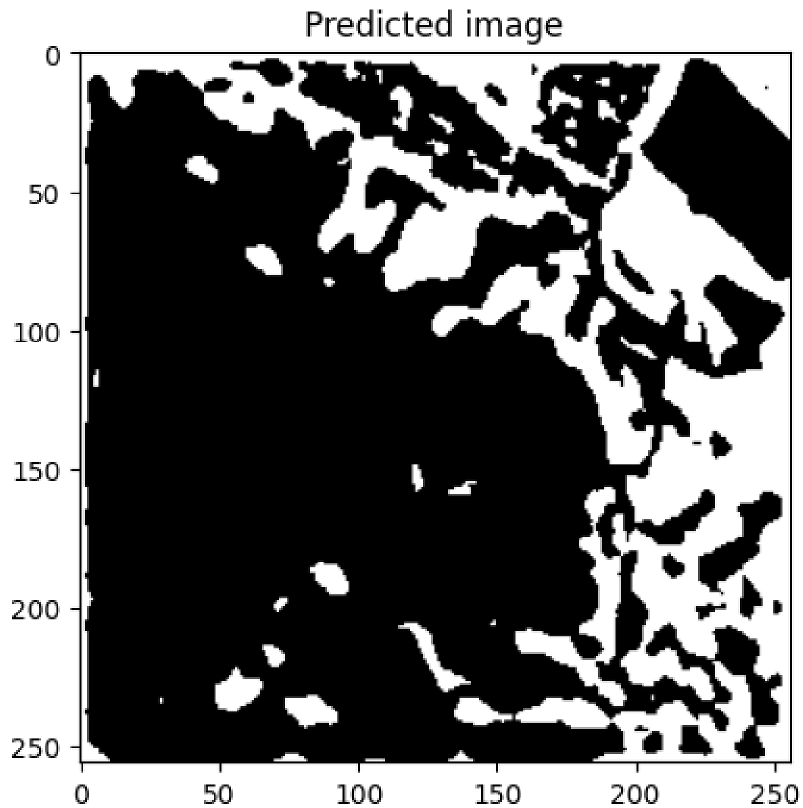
[**TODO**] Test the model on the test set and measure the accuracy.

[ ]:
```
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 [==============================] - 0s 203ms/step - loss: 0.6479 - accuracy:
0.7351
```




```
1/1 [==============================] - 0s 166ms/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 doing so we can
     # 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 can reduce the
     # 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.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: 0.7625885009765625
Image 2 - Accuracy: 0.7811737060546875
Image 3 - Accuracy: 0.8894500732421875
Image 4 - Accuracy: 0.642120361328125
Image 5 - Accuracy: 0.6657562255859375
Image 6 - Accuracy: 0.6487274169921875
Image 7 - Accuracy: 0.8273773193359375
Image 8 - Accuracy: 0.715728759765625
Image 9 - Accuracy: 0.8212127685546875
Image 10 - Accuracy: 0.63031005859375
Mean accuracy: 0.7384445190429687
1/1 [==============================] - 0s 56ms/step
```

[**TODO**] Finetune the Keras model using quantization-aware training and measure the accuracy on the test set after actually quantizing it

[ ]:
```
# the best way to quantize the parameter is through QAT (Quantization Aware␣
↪Training). This method consists in training the model to find the best␣
↪parameters that can be quantized in int8.
# So during the training the model will be aware that the parameters need to be␣
↪quantized, so it will find the best values to be quantized.
# A model quantized with QAT can perform similar to the original model, but␣
↪with a lower memory occupation. Note that QAT model need an ad-hoc training␣
↪and calibration.

# cannot apply QAT to batch normalizer layers
quant_aware_model = tfmot.quantization.keras.quantize_model(Unet_model)
quant_aware_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.
↪01),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
quant_aware_model.fit(X_train, Y_train, epochs=50)

converter = tf.lite.TFLiteConverter.from_keras_model(quant_aware_model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
quantized_aware_model = converter.convert()
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-30-7f43f456f466> in <cell line: 0>()
      4
      5 # cannot apply QAT to batch normalizer layers
----> 6 quant_aware_model = tfmot.quantization.keras.quantize_model(Unet_model)
      7 quant_aware_model.compile(optimizer=tf.keras.optimizers.
↪Adam(learning_rate=0.01),
```

```
      8                 loss='sparse_categorical_crossentropy',

/usr/local/lib/python3.11/dist-packages/tensorflow_model_optimization/python/core/quantization/keras/quantize.py in quantize_model(to_quantize, quantized_layer_name_prefix)
    139 
    140     annotated_model = quantize_annotate_model(to_quantize)
--> 141     return quantize_apply(
    142         annotated_model, quantized_layer_name_prefix=quantized_layer_name_prefix)
    143 

/usr/local/lib/python3.11/dist-packages/tensorflow_model_optimization/python/core/keras/metrics.py in inner(*args, **kwargs)
     72       except Exception as error:
     73         self.bool_gauge.get_cell(MonitorBoolGauge._FAILURE_LABEL).set(True)
---> 74         raise error
     75 
     76     if self.bool_gauge:

/usr/local/lib/python3.11/dist-packages/tensorflow_model_optimization/python/core/keras/metrics.py in inner(*args, **kwargs)
     67     def inner(*args, **kwargs):
     68       try:
---> 69         results = func(*args, **kwargs)
     70         self.bool_gauge.get_cell(MonitorBoolGauge._SUCCESS_LABEL).set(True)
     71         return results

/usr/local/lib/python3.11/dist-packages/tensorflow_model_optimization/python/core/quantization/keras/quantize.py in quantize_apply(model, scheme, quantized_layer_name_prefix)
    498   # `QuantizeConfig`.
    499 
--> 500   return keras.models.clone_model(
    501       transformed_model, input_tensors=None, clone_function=_quantize)
    502 

/usr/local/lib/python3.11/dist-packages/tf_keras/src/models/cloning.py in clone_model(model, input_tensors, clone_function)
    538                 clone_function or input_tensors
    539             ):
--> 540                 return _clone_functional_model(
    541                     model, input_tensors=input_tensors, layer_fn=clone_function
    542                 )
```

```
/usr/local/lib/python3.11/dist-packages/tf_keras/src/models/cloning.py in _clone_functional_model(model, input_tensors, layer_fn)
    216                 save_traces=False, in_tf_saved_model_scope=True
    217             ):
--> 218                 model_configs, created_layers = _clone_layers_and_model_config(
    219                     model, new_input_layers, layer_fn
    220                 )

/usr/local/lib/python3.11/dist-packages/tf_keras/src/models/cloning.py in _clone_layers_and_model_config(model, input_layers, layer_fn)
    296             return {}
    297
--> 298         config = functional.get_network_config(
    299             model, serialize_layer_fn=_copy_layer
    300         )

/usr/local/lib/python3.11/dist-packages/tf_keras/src/engine/functional.py in get_network_config(network, serialize_layer_fn, config)
   1590                 if isinstance(layer, Functional) and set_layers_legacy:
   1591                     layer.use_legacy_config = True
-> 1592                 layer_config = serialize_layer_fn(layer)
   1593                 layer_config["name"] = layer.name
   1594                 layer_config["inbound_nodes"] = filtered_inbound_nodes

/usr/local/lib/python3.11/dist-packages/tf_keras/src/models/cloning.py in _copy_layer(layer)
    293                 created_layers[layer.name] = InputLayer(**layer.get_config())
    294             else:
--> 295                 created_layers[layer.name] = layer_fn(layer)
    296             return {}
    297

/usr/local/lib/python3.11/dist-packages/tensorflow_model_optimization/python/core/quantization/keras/quantize.py in _quantize(layer)
    444               'instance to the `quantize_annotate_layer` '
    445               'API.')
--> 446         raise RuntimeError(
    447             error_msg.format(layer.name, layer.__class__,
    448                              quantize_registry.__class__))

RuntimeError: Layer batch_normalization_42:<class 'tf_keras.src.layers.normalization.batch_normalization.BatchNormalization'> is not supported. You can quantize this layer by passing a `tfmot.quantization.keras.QuantizeConfig` instance to the `quantize_annotate_layer` API.
```