Reto MeIA - sin aumentación del dataset

June 29, 2023

1 Semantic Segmentation Using HAGDAVS Dataset

1.1 Import Libraries

[ ]:
```
%%capture
!pip uninstall tensorflow -y
!pip uninstall tensorflow-io -y
!pip install tensorflow
!pip install --no-deps tensorflow-io
```

[ ]:
```
import tensorflow as tf
import tensorflow_io as tfio
from tensorflow import keras
from tensorflow.keras import layers
import os
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.callbacks import TensorBoard
import pathlib
from PIL import Image
import shutil
from tensorflow.keras.layers.experimental import preprocessing

from tensorflow.data import AUTOTUNE
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
```

1.2 Load Data

This code performs the following actions:

1. Defines directory paths for the input data, including the folder containing RGB images and the folder containing masks.
2. Sets the destination folder path to save the modified images.
3. Creates the new destination folder if it doesn't exist.
4. Retrieves the list of file names in the masks folder.
5. Iterates over each file in the list and does the following:
    - Checks if the file name contains the text "MClass".
    - If it does, modifies the file name by replacing "MClass" with "RGB".

- Creates an old file path and a new file path.
- Copies the old file to the new path.

In summary, the code takes masks in the "MClass" format and copies them to a new folder, but changes their names to the "RGB" format.

[ ]:
```
# Establecer rutas

platform_to_use='colab'
#platform_to_use='kaggle'

if platform_to_use=='colab':
  from google.colab import drive
  drive.mount('/content/drive', force_remount=True)
  shared_folder_path = "/content/drive/Shareddrives/MeIA/Modulo_3_Reto"  #␣
 ↪Reemplaza con la ruta (Ruta del dataset)
  output_path = f'{shared_folder_path}/Esteban_Test' # Ruta de resultados

elif platform_to_use=='kaggle':
  shared_folder_path = '/kaggle/input/hagdavs'
  output_path = '/kaggle/working/semantic-segmentation-hagdavs/HAGDAVS'

# Best model
filepath_best_model = f"{output_path}/weights.best.hdf5"
```

```
Mounted at /content/drive
```

[ ]:
```
data_dir = pathlib.Path(f"{shared_folder_path}/HAGDAVS")
images_dir = data_dir / 'RGB'
masks_dir = data_dir / 'MASK'

folder_path = masks_dir  # Ruta de la carpeta que contiene las imágenes
new_folder_path = f"{output_path}/MASK"

# Crear la nueva carpeta si no existe
os.makedirs(new_folder_path, exist_ok=True)

# Obtener la lista de nombres de archivo en la carpeta
file_list = os.listdir(folder_path)

# Aplicar la modificación del nombre y crear nuevos archivos en la nueva carpeta
for filename in file_list:
    if "MClass" in filename:
        new_filename = filename.replace("MClass", "RGB")
        old_path = os.path.join(folder_path, filename)
        new_path = os.path.join(new_folder_path, new_filename)
        shutil.copy2(old_path, new_path)
```

This code performs the following actions:

1. Defines directory paths for the image and mask folders.
2. Retrieves a list of file paths for the images and masks.
3. Creates empty lists to store image and mask patches.
4. Sets the desired patch size.
5. Defines a function to load and split the images and masks into patches.
   - It reads the image and mask files.
   - Removes the alpha channel from the images and masks.
   - Divides the image into patches based on the specified patch size.
   - Checks if all pixels in a mask patch are black, and if so, skips it.
   - Appends the image and mask patches to their respective lists.
6. Applies the load_and_split_patches function to each image and mask pair.
7. Converts the lists of patches into TensorFlow tensors.
8. Creates a dataset from the image and mask patches.

In summary, the code loads image and mask files, divides them into patches of a specified size, filters out patches with all-black masks, and creates a dataset for further processing or training.

[ ]:
```
# Ruta a las carpetas de imágenes y máscaras
ima_dir = pathlib.Path(data_dir)
images_dir = ima_dir / 'RGB'
m_dir = pathlib.Path(output_path)
mask_dir = m_dir / 'MASK'

# Obtener una lista de rutas de archivo para imágenes y máscaras
image_paths = sorted([str(path) for path in images_dir.glob('*.tif')])
mask_paths = sorted([str(path) for path in mask_dir.glob('*.tif')])

# Crear una lista para almacenar los parches de imágenes y máscaras
image_patches = []
mask_patches = []

# Tamaño del parche deseado
patch_size = (256, 256)


# Definir una función para cargar y dividir las imágenes y máscaras en parches
def load_and_split_patches(image_path, mask_path):
    image = tfio.experimental.image.decode_tiff(tf.io.read_file(image_path))
    mask = tfio.experimental.image.decode_tiff(tf.io.read_file(mask_path))

    # Eliminar el canal alfa de las imágenes y máscaras
    image = image[:, :, :3]
    mask = mask[:, :, :3]

    # Dividir la imagen en parches
    for i in range(0, image.shape[0], patch_size[0]):
```

```
        for j in range(0, image.shape[1], patch_size[1]):
            patch_image = image[i:i+patch_size[0], j:j+patch_size[1], :]
            patch_mask = mask[i:i+patch_size[0], j:j+patch_size[1], :]

            # Verificar si todos los píxeles en el parche de la máscara son␣
 ↪negros
            if tf.reduce_all(tf.math.equal(patch_mask, [0, 0, 0])):
                continue

            #patch_mask = convertir_colores(patch_mask)
            image_patches.append(patch_image)
            mask_patches.append(patch_mask)

    return None

# Aplicar la función de carga y división de parches a cada par de rutas de␣
 ↪archivo
for image_path, mask_path in zip(image_paths, mask_paths):
    load_and_split_patches(image_path, mask_path)

# Convertir las listas de parches en tensores
image_patches = tf.convert_to_tensor(image_patches)
mask_patches = tf.convert_to_tensor(mask_patches)

# Crear un dataset a partir de los parches de imágenes y máscaras
dataset = tf.data.Dataset.from_tensor_slices((image_patches, mask_patches))
```

The size of images in dataset:

[ ]:
```
dataset
```

[ ]:
```
<_TensorSliceDataset element_spec=(TensorSpec(shape=(256, 256, 3),
dtype=tf.uint8, name=None), TensorSpec(shape=(256, 256, 3), dtype=tf.uint8,
name=None))>
```

1.3 Display Data

Function for show the image, mask and prediction mask

[ ]:
```
def display(display_list):
  plt.figure(figsize=(15, 15))

  title = ["Input Image", "True Mask", "Predicted Mask"]

  for i in range(len(display_list)):
    plt.subplot(1, len(display_list), i+1)
    plt.title(title[i])
```

```
    plt.imshow(tf.keras.utils.array_to_img(display_list[i]))
    plt.axis("off")
  plt.show()
```

1.4 Using only one class

This code defines a function called `convertir_mascara` and applies it to a dataset using the `map` function.

The `convertir_mascara` function performs the following actions:

1. Casts the input mask tensor to `float32` data type.
2. Creates a new mask tensor filled with zeros, with dimensions `(256, 256, 1)`.
3. Assigns values corresponding to each class in the mask:
   - If all RGB values in the mask are `[0, 0, 0]`, assigns `0.0` to the corresponding pixel in the converted mask.
   - If all RGB values are `[255, 0, 0]`, assigns `0.0`.
   - If all RGB values are `[0, 255, 0]`, assigns `1.0`.
   - If all RGB values are `[0, 0, 255]`, assigns `0.0`.
4. Returns the converted mask.

The code then applies the `convertir_mascara` function to each element in the `dataset` using the `map` function. The `map` function takes a lambda function that applies the conversion function to each `(image, mask)` pair in the dataset, resulting in a new dataset named `mapped_dataset`. The images in the dataset remain unchanged, while the masks are converted using the `convertir_mascara` function.

[ ]:
```
def convertir_mascara(mascara):
    mascara = tf.cast(mascara, dtype=tf.float32)
    mascara_convertida = tf.zeros((256, 256, 1), dtype=tf.float32)

    # Asigna valores correspondientes a cada clase
    mascara_convertida = tf.where(tf.reduce_all(tf.equal(mascara, [0, 0, 0]),␣
↪axis=-1, keepdims=True), 0.0, mascara_convertida)
    mascara_convertida = tf.where(tf.reduce_all(tf.equal(mascara, [255, 0, 0]),␣
↪axis=-1, keepdims=True), 0.0, mascara_convertida)
    mascara_convertida = tf.where(tf.reduce_all(tf.equal(mascara, [0, 255, 0]),␣
↪axis=-1, keepdims=True), 1.0, mascara_convertida)
    mascara_convertida = tf.where(tf.reduce_all(tf.equal(mascara, [0, 0, 255]),␣
↪axis=-1, keepdims=True), 0.0, mascara_convertida)

    return mascara_convertida

mapped_dataset = dataset.map(lambda x, y: (x, convertir_mascara(y)))
```

This code defines a function called `filter_func` that filters the `mapped_dataset` based on certain criteria using the `filter` function.

The `filter_func` function performs the following actions:

1. Reshapes the mask tensor into a 1-dimensional tensor.
2. Uses `tf.unique` to obtain the unique classes present in the mask.
3. Checks if the number of unique classes (`tf.size(unique_classes)`) is greater than or equal to 2.
    - If there are two or more unique classes, it returns `True`, indicating that the image and mask pair should be included in the filtered dataset.
    - If there are fewer than two unique classes, it returns `False`, indicating that the image and mask pair should be filtered out.

The code then applies the `filter_func` function to each element in the `mapped_dataset` using the `filter` function. The `filter` function takes the lambda function `filter_func` as an argument and returns a new dataset named `filtered_dataset` that contains only the image and mask pairs that satisfy the filtering criteria.

[ ]:
```
def filter_func(image, mask):
    unique_classes = tf.unique(tf.reshape(mask, [-1]))[0]
    return tf.size(unique_classes) >= 2

filtered_dataset = mapped_dataset.filter(filter_func)

dataset_length = 0
for _ in filtered_dataset:
    dataset_length += 1

print("Longitud aproximada del dataset filtrado:", dataset_length)
```

```
Longitud aproximada del dataset filtrado: 1043
```

1.5 Splitting data

train/test/val split

[ ]:
```
# dividir el dataset en conjuntos de entrenamiento, validación y prueba
total_samples = (dataset_length)
train_size = int(0.7 * total_samples)
val_size = int(0.15 * total_samples)
test_size = total_samples - train_size - val_size

train_dataset = filtered_dataset.take(train_size)
val_dataset = filtered_dataset.skip(train_size).take(val_size)
test_dataset = filtered_dataset.skip(train_size + val_size).take(test_size)
```

[ ]:
```
BATCH_SIZE = 16
BUFFER_SIZE = 100
```

[ ]:
```
train_batches = train_dataset.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).
 ↪repeat()
train_batches = train_batches.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
```

```
validation_batches = test_dataset.batch(BATCH_SIZE)
test_batches = test_dataset.batch(BATCH_SIZE)
```

1.6 Unet

[ ]:
```
# Se cambió la función de activación relu por LeakyReLU dado que daba mejores␣
↪resultados
# Funcion normal con maximo de 1024 neuronas
# Nota: solo correr una red

def double_conv_block(x, n_filters):

    # Conv2D then ReLU activation
    x = layers.Conv2D(n_filters, 3, padding = "same", kernel_initializer =␣
↪"he_normal")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("LeakyReLU")(x)
    # Conv2D then ReLU activation
    x = layers.Conv2D(n_filters, 3, padding = "same", kernel_initializer =␣
↪"he_normal")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("LeakyReLU")(x)
    # Conv2D then ReLU activation
    x = layers.Conv2D(n_filters, 3, padding = "same", kernel_initializer =␣
↪"he_normal")(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("LeakyReLU")(x)
    # dropout
    x = layers.Dropout(0.2)(x)

    return x

def downsample_block(x, n_filters):
    f = double_conv_block(x, n_filters)
    p = layers.MaxPool2D(2)(f)
    p = layers.Dropout(0.2)(p)

    return f, p

def upsample_block(x, conv_features, n_filters):
    # upsample
    x = layers.Conv2DTranspose(n_filters, 3, 2, padding="same")(x)
    # concatenate
    x = layers.concatenate([x, conv_features])
    # dropout
    x = layers.Dropout(0.2)(x)
```

```
    # Conv2D twice with ReLU activation
    x = double_conv_block(x, n_filters)

    return x

def build_unet_model():

    # inputs
    inputs = layers.Input(shape=(256,256,3))

    # encoder: contracting path - downsample
    # 1 - downsample
    f1, p1 = downsample_block(inputs, 64)
    # 2 - downsample
    f2, p2 = downsample_block(p1, 128)
    # 3 - downsample
    f3, p3 = downsample_block(p2, 256)
    # 4 - downsample
    f4, p4 = downsample_block(p3, 512)

    # 5 - bottleneck
    bottleneck = double_conv_block(p4, 1024)
    bottleneck = layers.Dropout(0.3)(bottleneck)

    # decoder: expanding path - upsample
    # 6 - upsample
    u6 = upsample_block(bottleneck, f4, 512)
    # 7 - upsample
    u7 = upsample_block(u6, f3, 256)
    # 8 - upsample
    u8 = upsample_block(u7, f2, 128)
    # 9 - upsample
    u9 = upsample_block(u8, f1, 64)

    # outputs
    outputs = layers.Conv2D(1, 1, padding="same", activation = "sigmoid")(u9)

    # unet model with Keras Functional API
    unet_model = tf.keras.Model(inputs, outputs, name="U-Net")

    return unet_model

unet_model = build_unet_model()

#Función de pérdida para clasificación binaria: BinaryFocalCrossentropy

BFC = tf.keras.losses.BinaryFocalCrossentropy(
```

```
    apply_class_balancing=True,    #Balanceo de datos
    alpha=0.25,    #Parámetro de peso
    gamma=2.0,     #Focussing parameter
    from_logits=False,    #False para datos con función sigmoide en la última capa
    label_smoothing=0.0,
    axis=-1,
    reduction=tf.keras.losses.Reduction.AUTO, #AUTO según el modelo
    )

#loss = keras.losses.sparse_categorical_crossentropy()

unet_model.compile(optimizer=tf.keras.optimizers.Adam(),
                   loss=BFC,
                   metrics=["accuracy"])
```

[ ]:
```
# Se cambió la función de activación relu por LeakyReLU dado que daba mejores␣
 ↪resultados, agregue un parametro alpha
# Funcion con modelo + una capa con maximo de 2048 neuronas
# Nota: solo correr una red

import numpy as np
from sklearn.model_selection import KFold, cross_val_score
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

def double_conv_block(x, n_filters):
    # Conv2D then LeakyReLU
    x = layers.Conv2D(n_filters, 3, padding="same",␣
 ↪kernel_initializer="he_normal")(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(alpha=0.2)(x)
    # Conv2D then LeakyReLU
    x = layers.Conv2D(n_filters, 3, padding="same",␣
 ↪kernel_initializer="he_normal")(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(alpha=0.2)(x)
    # Conv2D then LeakyReLU
    x = layers.Conv2D(n_filters, 3, padding="same",␣
 ↪kernel_initializer="he_normal")(x)
    x = layers.BatchNormalization()(x)
    x = layers.LeakyReLU(alpha=0.2)(x)
    # dropout
    x = layers.Dropout(0.2)(x)

    return x

def downsample_block(x, n_filters):
    f = double_conv_block(x, n_filters)
```

```
    p = layers.MaxPool2D(2)(f)
    p = layers.Dropout(0.2)(p)

    return f, p

def upsample_block(x, conv_features, n_filters):
    # upsample
    x = layers.Conv2DTranspose(n_filters, 3, 2, padding="same")(x)
    # concatenate
    x = layers.concatenate([x, conv_features])
    # dropout
    x = layers.Dropout(0.2)(x)
    # Conv2D twice with ReLU activation
    x = double_conv_block(x, n_filters)

    return x

def build_unet_model():

    # inputs
    inputs = layers.Input(shape=(256,256,3))
    # encoder: contracting path - downsample
    # 1 - downsample
    f1, p1 = downsample_block(inputs, 64)
    # 2 - downsample
    f2, p2 = downsample_block(p1, 128)
    # 3 - downsample
    f3, p3 = downsample_block(p2, 256)
    # 4 - downsample
    f4, p4 = downsample_block(p3, 512)
    # 5 - downsample
    f5, p5 = downsample_block(p4, 1024)

    # 6 - bottleneck
    bottleneck = double_conv_block(p5, 2048)
    bottleneck = layers.Dropout(0.2)(bottleneck)

    # decoder: expanding path - upsample
    # 7 - upsample
    u7 = upsample_block(bottleneck, f5, 1024)
    # 8 - upsample
    u8 = upsample_block(u7, f4, 512)
    # 9 - upsample
    u9 = upsample_block(u8, f3, 256)
    # 10 - upsample
    u10 = upsample_block(u9, f2, 128)
    # 11 - upsample
```

```
    u11 = upsample_block(u10, f1, 64)

    # outputs
    outputs = layers.Conv2D(1, 1, padding="same", activation = "sigmoid")(u11)

    # unet model with Keras Functional API
    unet_model = tf.keras.Model(inputs, outputs, name="U-Net")

    return unet_model

unet_model = build_unet_model()

#loss = keras.losses.sparse_categorical_crossentropy()

unet_model.compile(optimizer=tf.keras.optimizers.Adam(),
                   loss='binary_crossentropy',
                   metrics=["accuracy"])
```

1.7 Training

[ ]:
```
keras.backend.clear_session()
NUM_EPOCHS = 10
PATIENCE = 5 # Stop if not improve in PATIENCE epoch
BATCH_SIZE = 16

# To callbacks
# Save in filepath_best_model the best model according with validation accuracy
# and stop after patience epoch if this do not improve. The risk is that
# loss validation data is not considerate.


checkpoint = ModelCheckpoint(filepath_best_model, monitor='val_accuracy',␣
↪verbose=1, save_best_only=True, mode='max')
es = EarlyStopping(monitor='val_accuracy', patience=PATIENCE)
callbacks_list = [checkpoint, es]

STEPS_PER_EPOCH = total_samples // BATCH_SIZE

VAL_SUBSPLITS = 5
VALIDATION_STEPS = test_size // BATCH_SIZE // VAL_SUBSPLITS

model_history = unet_model.fit(train_batches,
                               callbacks=callbacks_list,
                               epochs=NUM_EPOCHS,
                               steps_per_epoch=STEPS_PER_EPOCH,
                               validation_steps=VALIDATION_STEPS,
                               validation_data=validation_batches,)
```

```
Epoch 1/10
65/65 [==============================] - ETA: 0s - loss: 0.0090 - accuracy:
0.9680
Epoch 1: val_accuracy improved from -inf to 0.94593, saving model to
/content/drive/Shareddrives/MeIA/Modulo_3_Reto/Esteban_Test/weights.best.hdf5
65/65 [==============================] - 135s 2s/step - loss: 0.0090 - accuracy:
0.9680 - val_loss: 0.1891 - val_accuracy: 0.9459
Epoch 2/10
65/65 [==============================] - ETA: 0s - loss: 0.0090 - accuracy:
0.9653
Epoch 2: val_accuracy did not improve from 0.94593
65/65 [==============================] - 123s 2s/step - loss: 0.0090 - accuracy:
0.9653 - val_loss: 0.1640 - val_accuracy: 0.8196
Epoch 3/10
65/65 [==============================] - ETA: 0s - loss: 0.0086 - accuracy:
0.9668
Epoch 3: val_accuracy did not improve from 0.94593
65/65 [==============================] - 123s 2s/step - loss: 0.0086 - accuracy:
0.9668 - val_loss: 0.0304 - val_accuracy: 0.8476
```

```
Epoch 4/10
65/65 [==============================] - ETA: 0s - loss: 0.0080 - accuracy:
0.9689
Epoch 4: val_accuracy did not improve from 0.94593
65/65 [==============================] - 123s 2s/step - loss: 0.0080 - accuracy:
0.9689 - val_loss: 0.0154 - val_accuracy: 0.9412
Epoch 5/10
65/65 [==============================] - ETA: 0s - loss: 0.0081 - accuracy:
0.9682
Epoch 5: val_accuracy improved from 0.94593 to 0.95360, saving model to
/content/drive/Shareddrives/MeIA/Modulo_3_Reto/Esteban_Test/weights.best.hdf5
65/65 [==============================] - 149s 2s/step - loss: 0.0081 - accuracy:
0.9682 - val_loss: 0.0144 - val_accuracy: 0.9536
Epoch 6/10
65/65 [==============================] - ETA: 0s - loss: 0.0069 - accuracy:
0.9731
Epoch 6: val_accuracy improved from 0.95360 to 0.95619, saving model to
/content/drive/Shareddrives/MeIA/Modulo_3_Reto/Esteban_Test/weights.best.hdf5
65/65 [==============================] - 135s 2s/step - loss: 0.0069 - accuracy:
0.9731 - val_loss: 0.0173 - val_accuracy: 0.9562
Epoch 7/10
65/65 [==============================] - ETA: 0s - loss: 0.0078 - accuracy:
0.9699
Epoch 7: val_accuracy improved from 0.95619 to 0.96245, saving model to
/content/drive/Shareddrives/MeIA/Modulo_3_Reto/Esteban_Test/weights.best.hdf5
65/65 [==============================] - 143s 2s/step - loss: 0.0078 - accuracy:
0.9699 - val_loss: 0.0103 - val_accuracy: 0.9625
Epoch 8/10
65/65 [==============================] - ETA: 0s - loss: 0.0069 - accuracy:
0.9732
Epoch 8: val_accuracy did not improve from 0.96245
65/65 [==============================] - 123s 2s/step - loss: 0.0069 - accuracy:
0.9732 - val_loss: 0.0141 - val_accuracy: 0.9596
Epoch 9/10
65/65 [==============================] - ETA: 0s - loss: 0.0067 - accuracy:
0.9742
Epoch 9: val_accuracy improved from 0.96245 to 0.96350, saving model to
/content/drive/Shareddrives/MeIA/Modulo_3_Reto/Esteban_Test/weights.best.hdf5
65/65 [==============================] - 138s 2s/step - loss: 0.0067 - accuracy:
0.9742 - val_loss: 0.0129 - val_accuracy: 0.9635
Epoch 10/10
65/65 [==============================] - ETA: 0s - loss: 0.0068 - accuracy:
0.9744
Epoch 10: val_accuracy did not improve from 0.96350
65/65 [==============================] - 123s 2s/step - loss: 0.0068 - accuracy:
0.9744 - val_loss: 0.0192 - val_accuracy: 0.9509
```

1.8 Load Model

```
[ ]: # Create a basic model instance
model_loaded = build_unet_model()

# Loads the weights
model_loaded.load_weights(filepath_best_model)
```

1.9 Visualization of predictions

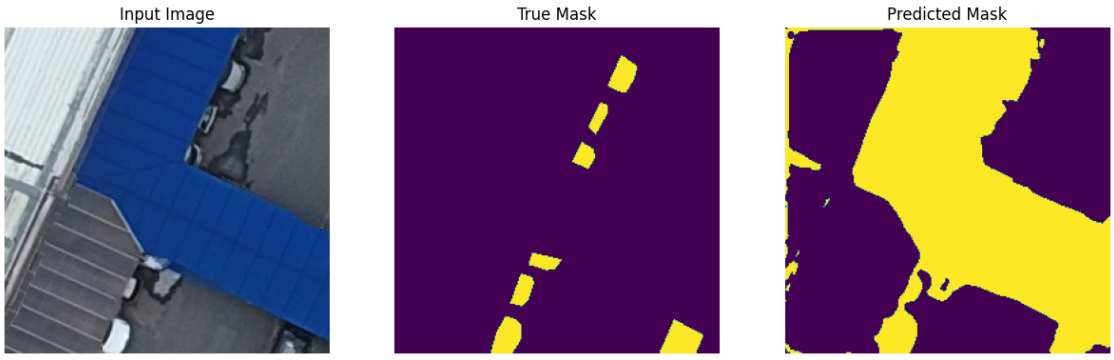
```
[ ]: def create_mask(pred_mask, threshold):
  pred_mask = tf.cast(pred_mask, dtype=tf.float32)  # convertir a float
  pred_mask = tf.where(pred_mask > threshold, 1.0, 0.0)  # usar float para el␣
 ↪threshold
  return pred_mask[0]

def show_predictions(dataset, model, num, threshold=0.2):
  if dataset:
    for image, mask in dataset.take(num):
      pred_mask = model.predict(image)
      display([image[0], mask[0], create_mask(pred_mask,threshold)])
  else:
    display([sample_image, sample_mask,
             create_mask(model.predict(sample_image[tf.newaxis, ...]))])
```

1.9.1 Observations

We obtain different results when threshold have several values.

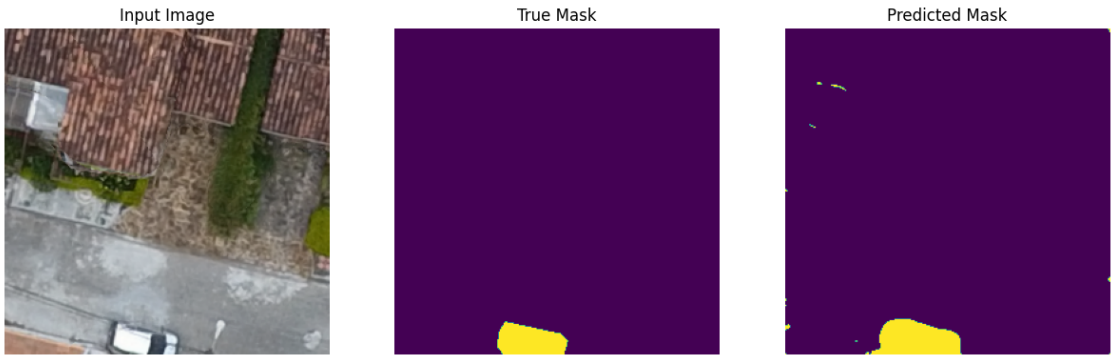
```
[ ]: show_predictions(test_batches, model_loaded, 16, 0.15)
```

```
1/1 [==============================] - 1s 561ms/step
```

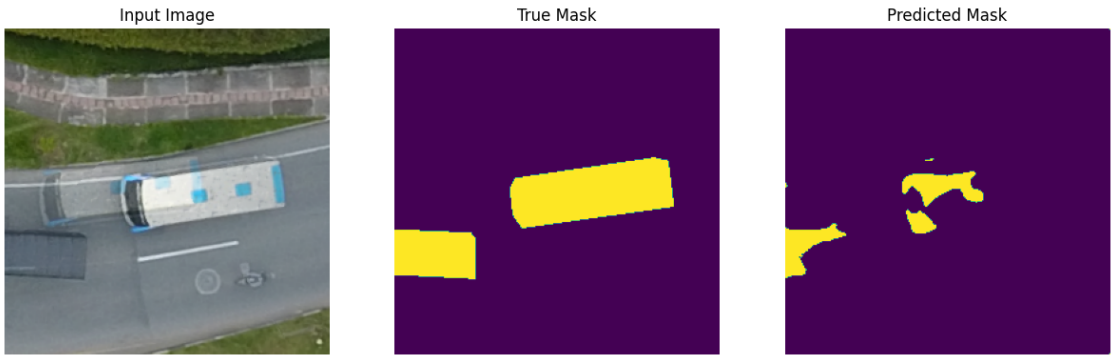



```
1/1 [==============================] - 0s 52ms/step
```

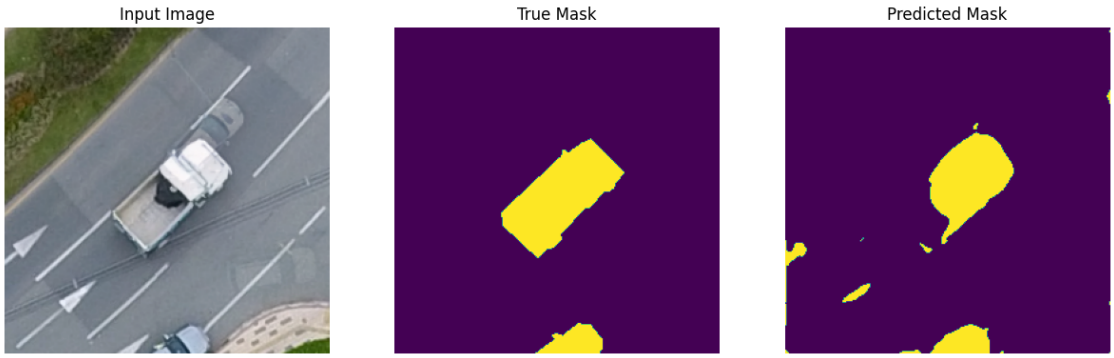

```
1/1 [==============================] - 0s 32ms/step
```



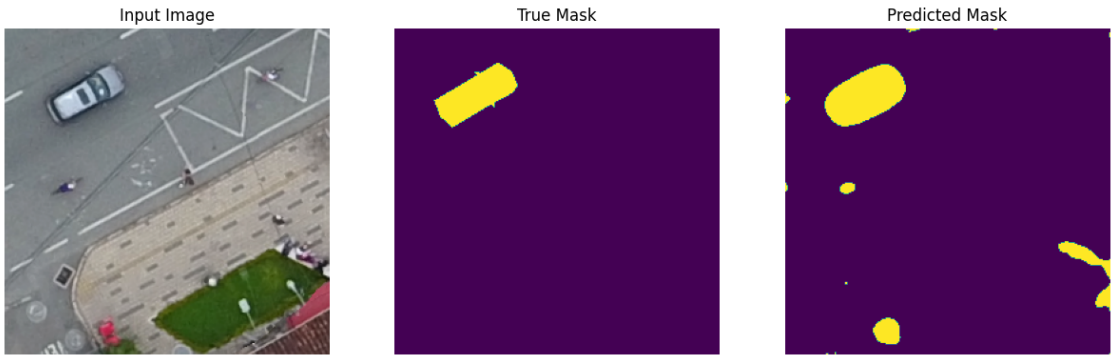

```
1/1 [==============================] - 0s 61ms/step
```



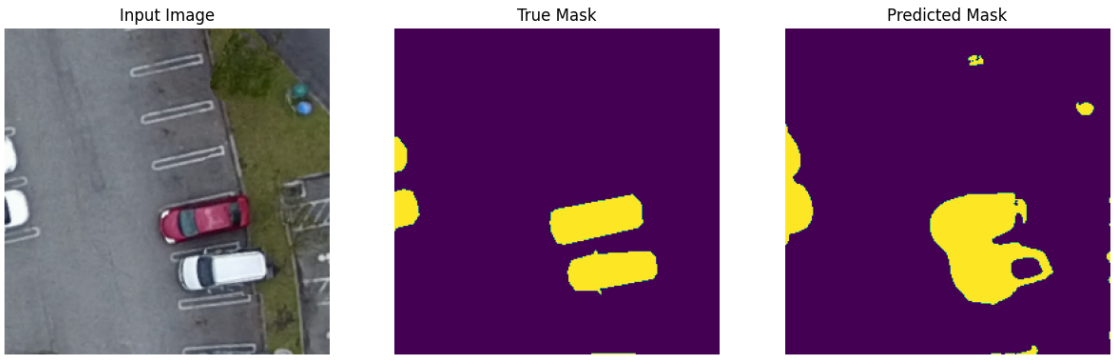

```
1/1 [==============================] - 0s 70ms/step
```



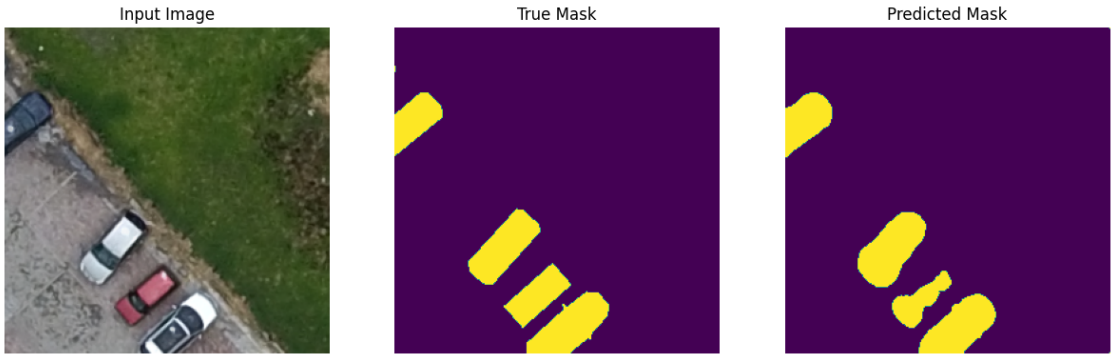

```
1/1 [==============================] - 0s 104ms/step
```



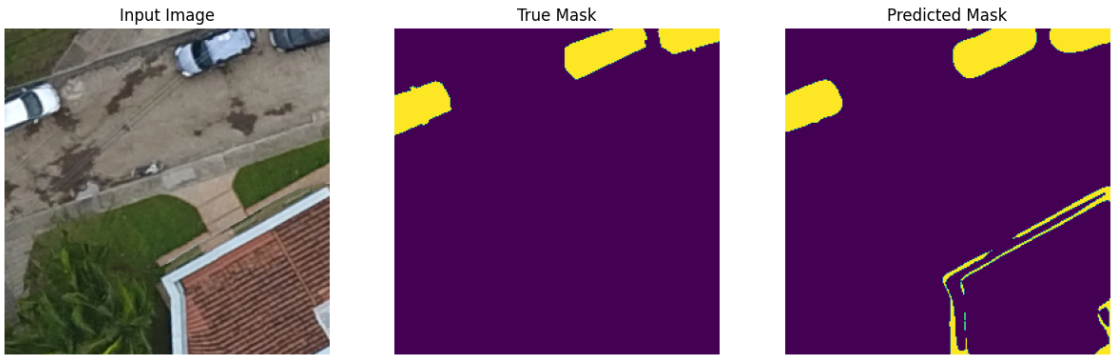

```
1/1 [==============================] - 0s 93ms/step
```



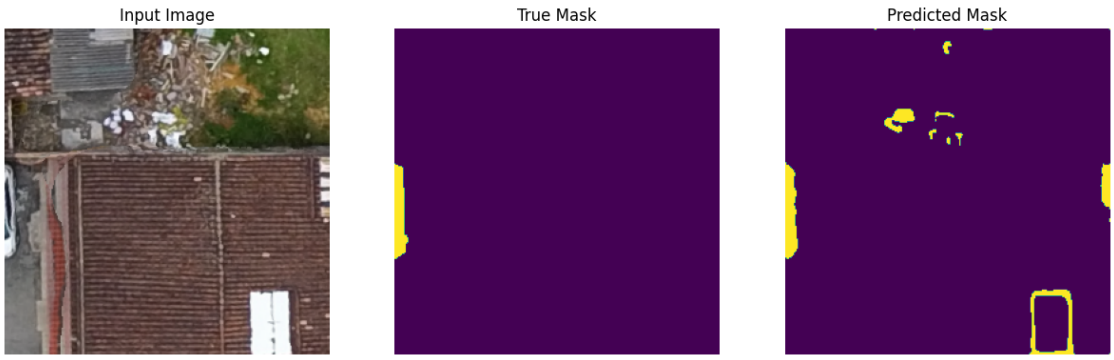

```
1/1 [==============================] - 0s 106ms/step
```




```
1/1 [==============================] - 0s 94ms/step
```

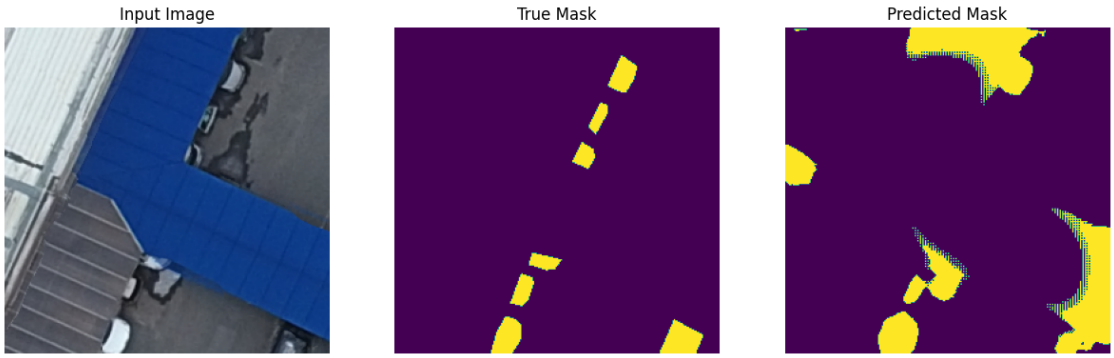



```
1/1 [==============================] - 0s 47ms/step
```



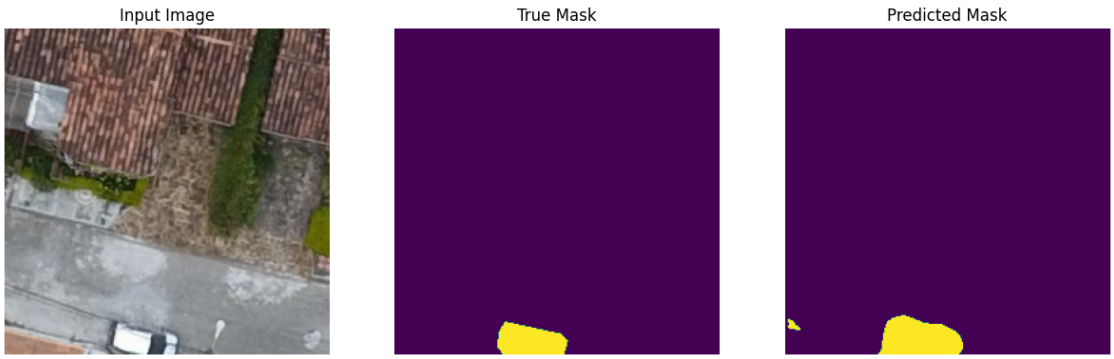
[ ]:
```
show_predictions(test_batches, model_loaded, 16, 0.2)
```

```
1/1 [==============================] - 0s 37ms/step
```

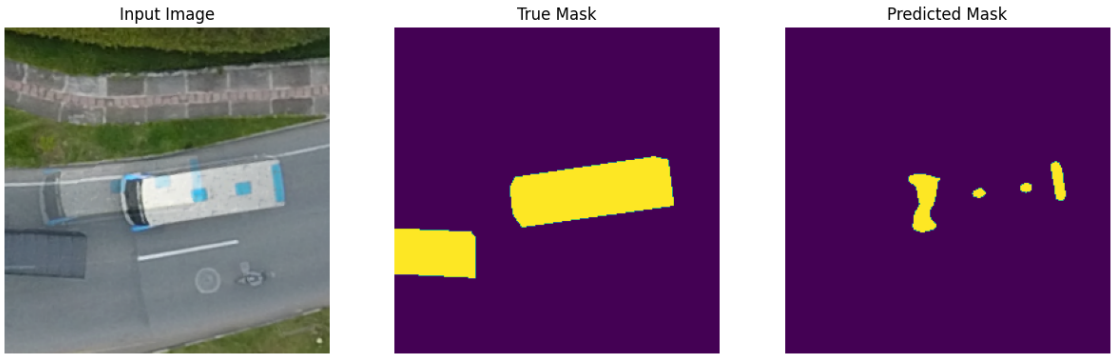



```
1/1 [==============================] - 0s 35ms/step
```

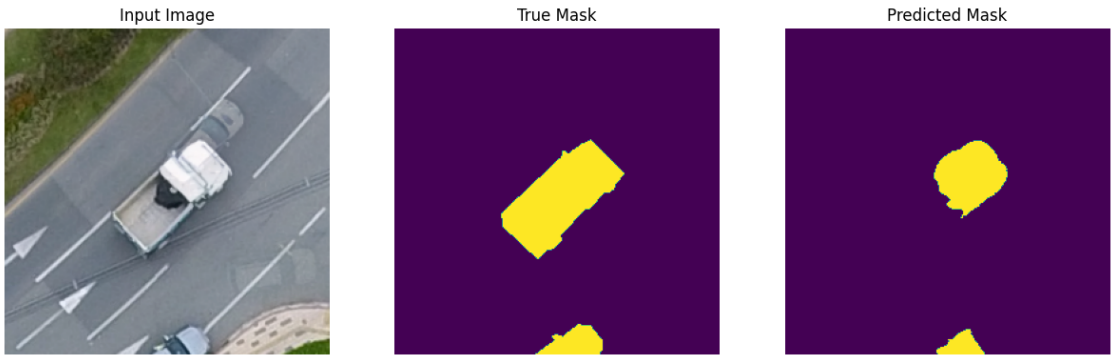

```
1/1 [==============================] - 0s 41ms/step
```



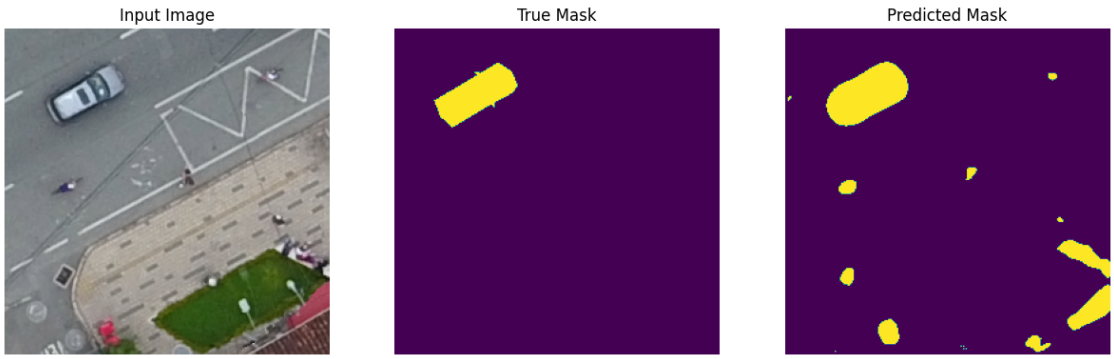

```
1/1 [==============================] - 0s 40ms/step
```



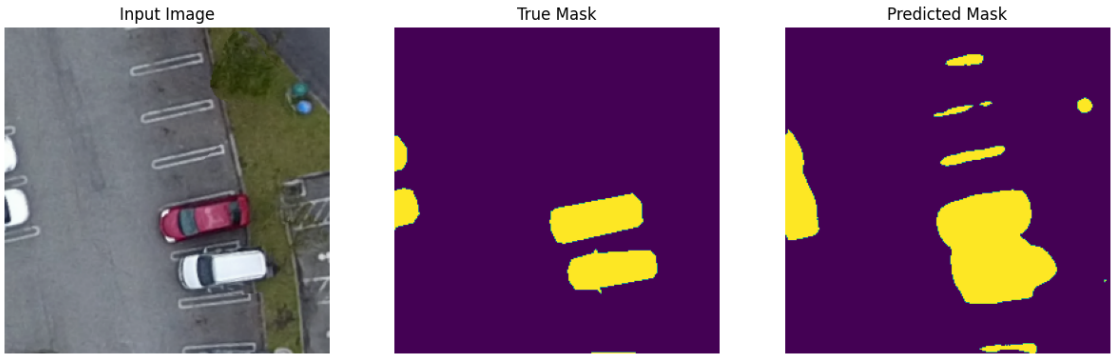

```
1/1 [==============================] - 0s 57ms/step
```



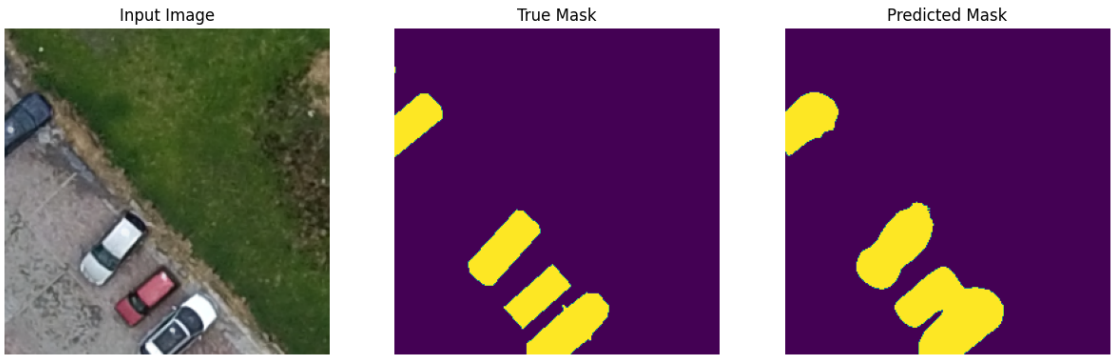

```
1/1 [==============================] - 0s 50ms/step
```



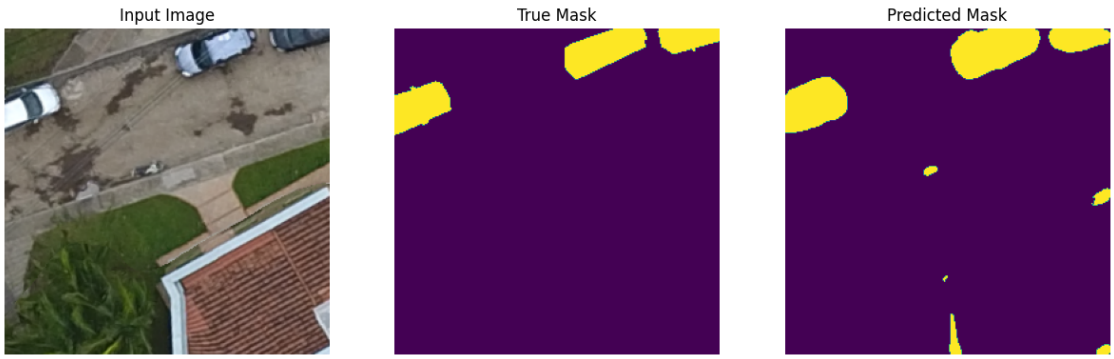

```
1/1 [==============================] - 0s 63ms/step
```



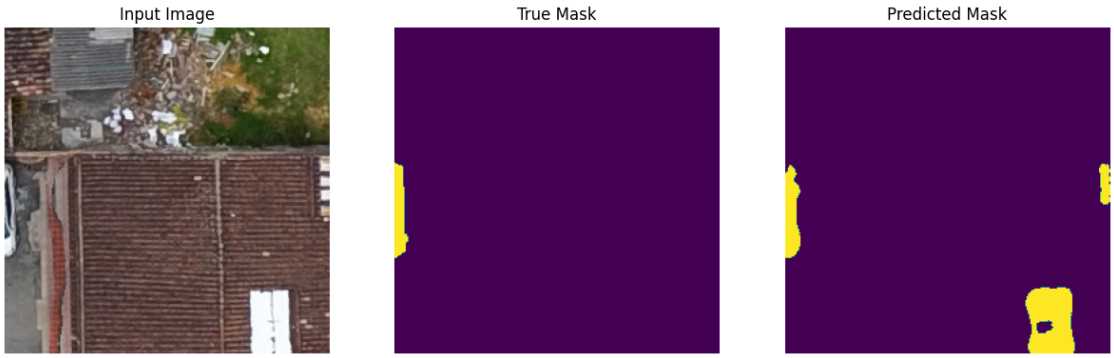

```
1/1 [==============================] - 0s 116ms/step
```




```
1/1 [==============================] - 0s 62ms/step
```

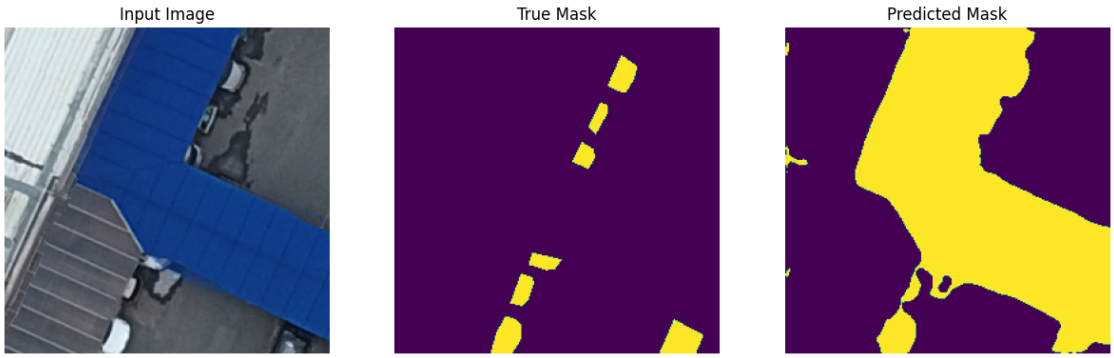



```
1/1 [==============================] - 9s 9s/step
```



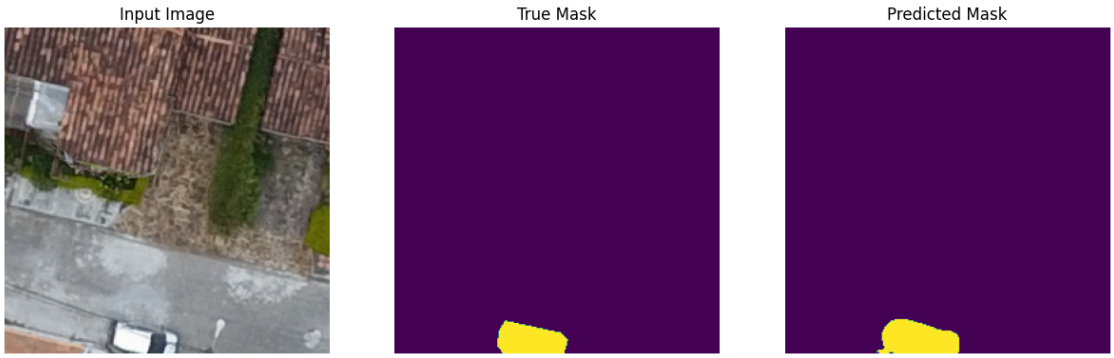
[ ]:
```
show_predictions(test_batches, model_loaded, 16, 0.20)
```

```
1/1 [==============================] - 0s 101ms/step
```

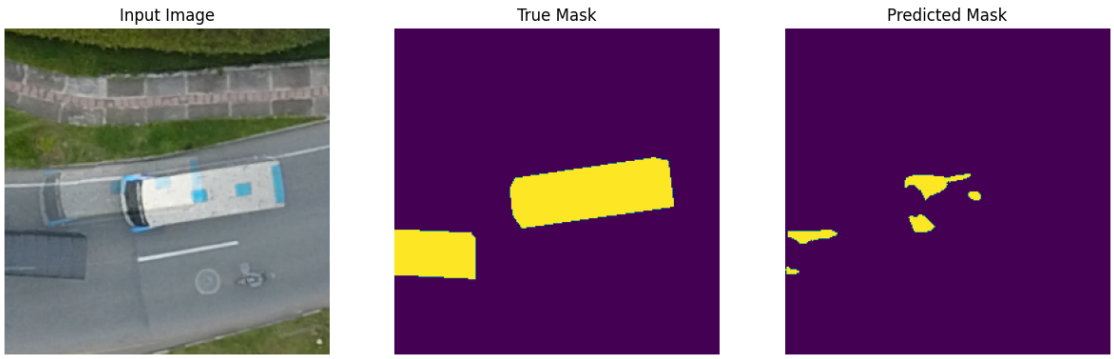



```
1/1 [==============================] - 0s 64ms/step
```

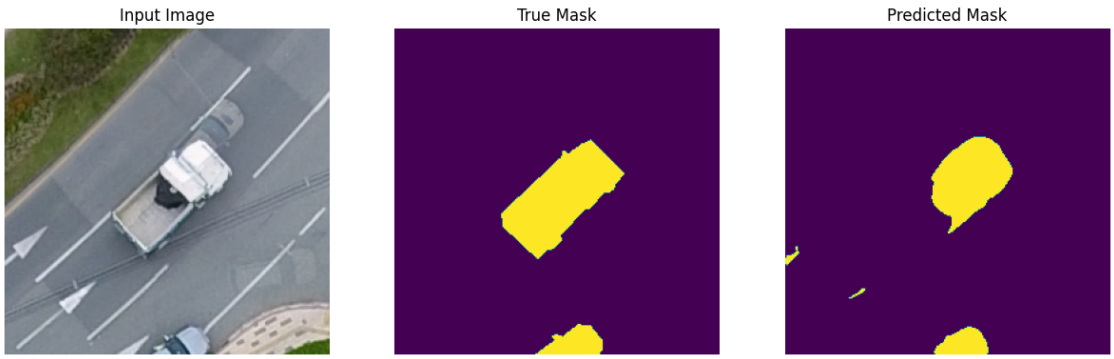

```
1/1 [==============================] - 0s 84ms/step
```



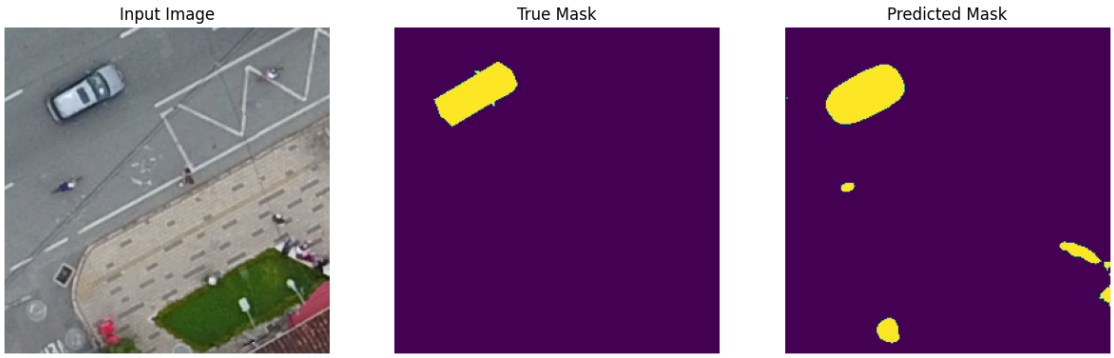

```
1/1 [==============================] - 0s 65ms/step
```



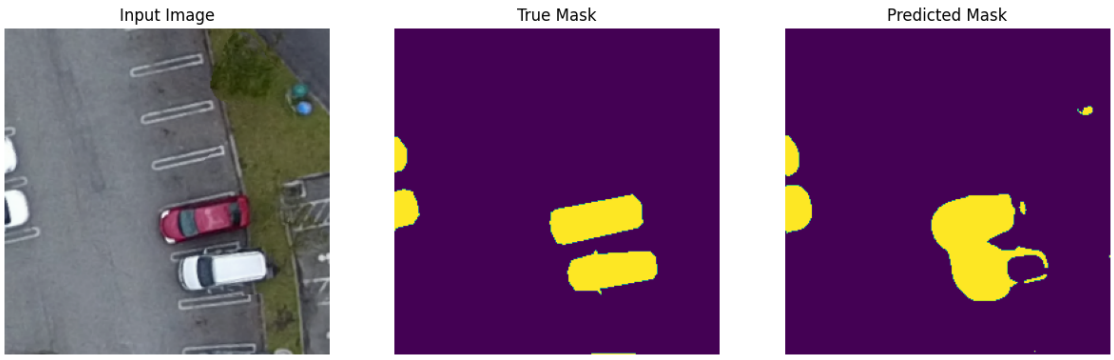

```
1/1 [==============================] - 0s 54ms/step
```



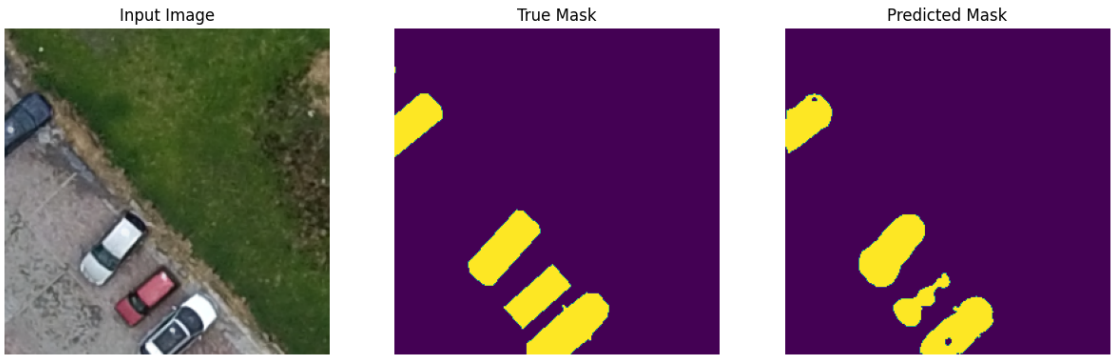

```
1/1 [==============================] - 0s 42ms/step
```



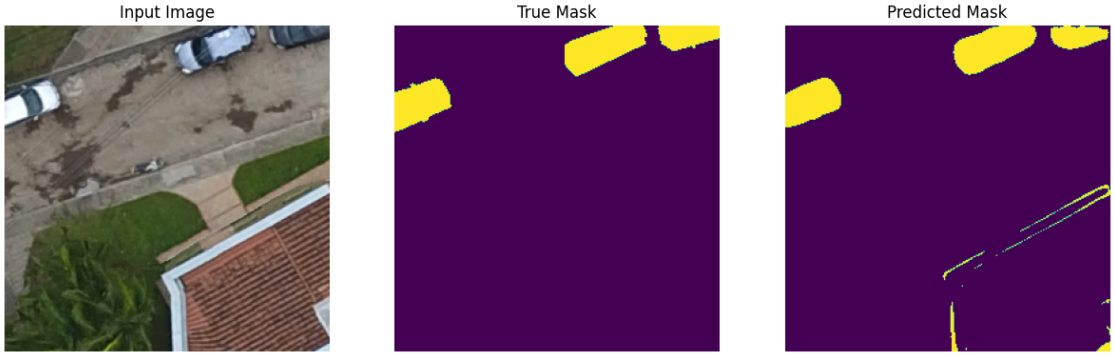

```
1/1 [==============================] - 0s 75ms/step
```



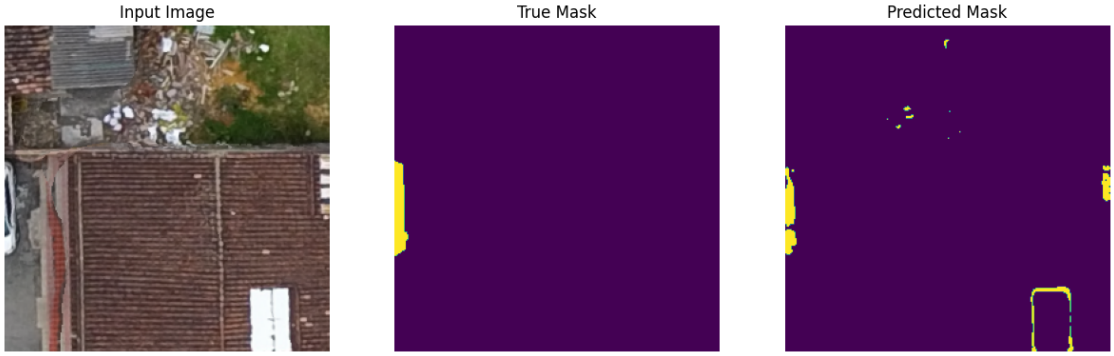

```
1/1 [==============================] - 0s 51ms/step
```




```
1/1 [==============================] - 0s 48ms/step
```

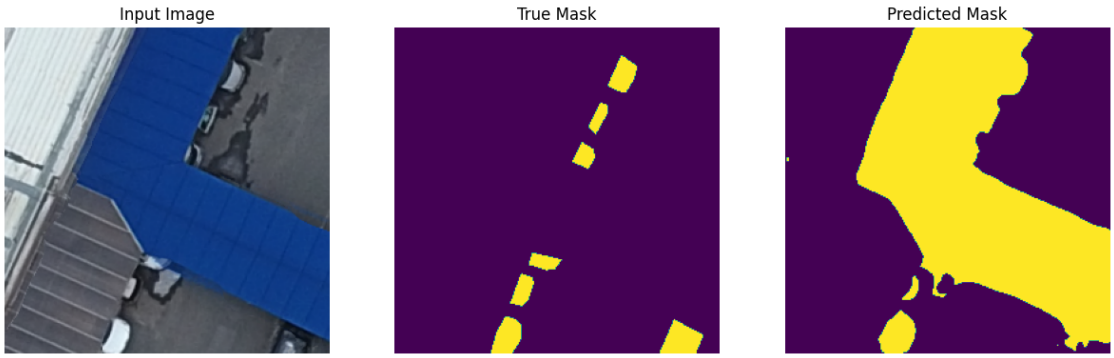



```
1/1 [==============================] - 0s 41ms/step
```



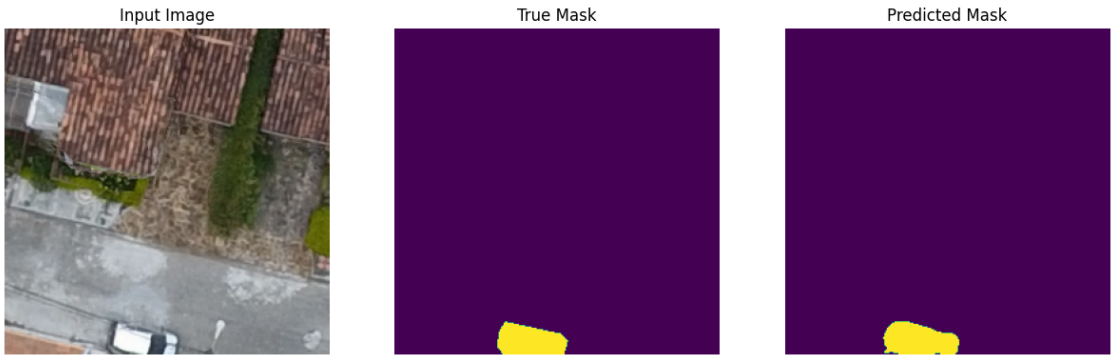
[ ]:
```
show_predictions(test_batches, model_loaded, 16, 0.25)
```

```
1/1 [==============================] - 0s 42ms/step
```

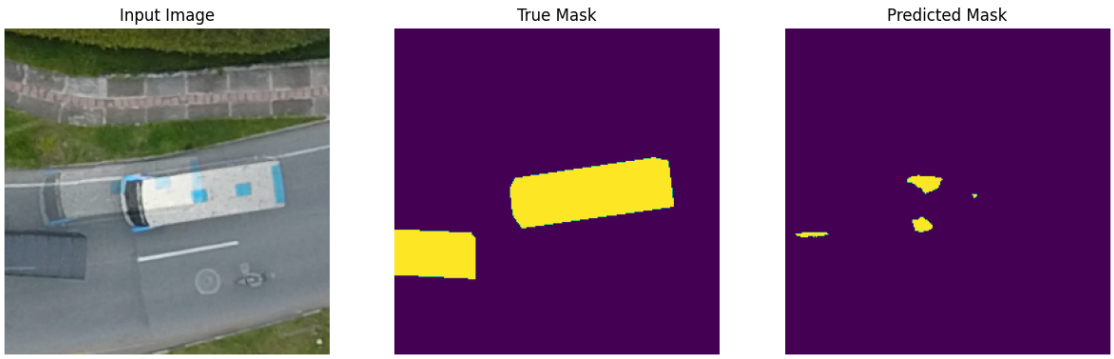



```
1/1 [==============================] - 0s 35ms/step
```

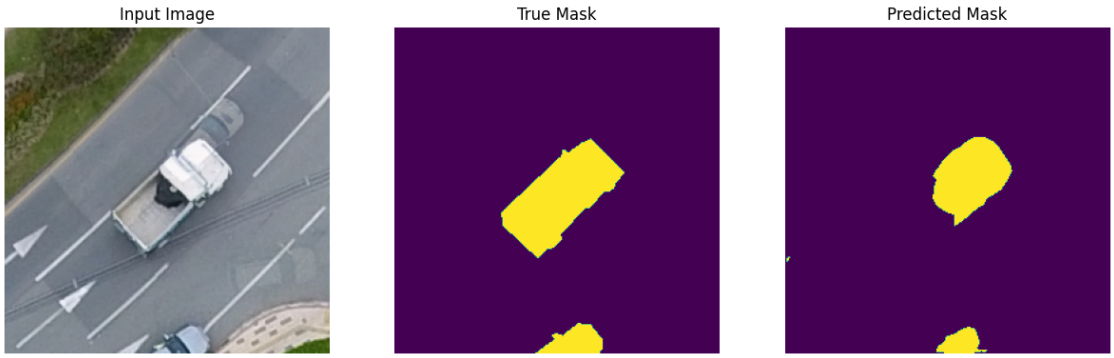

```
1/1 [==============================] - 0s 32ms/step
```



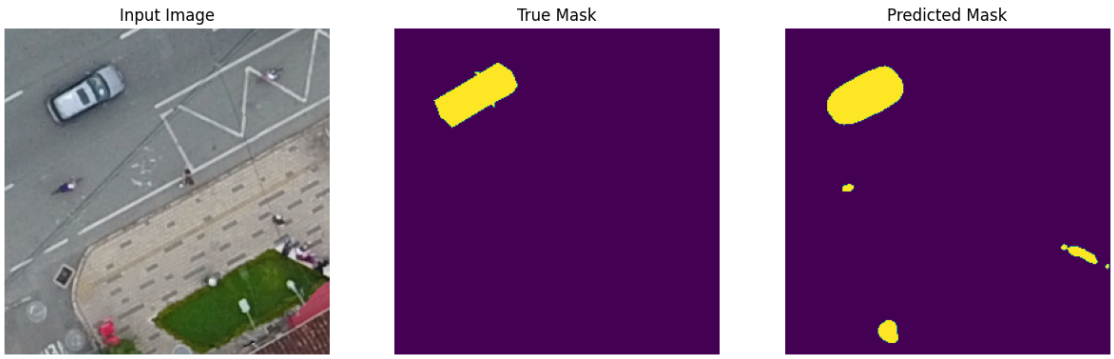

```
1/1 [==============================] - 0s 66ms/step
```



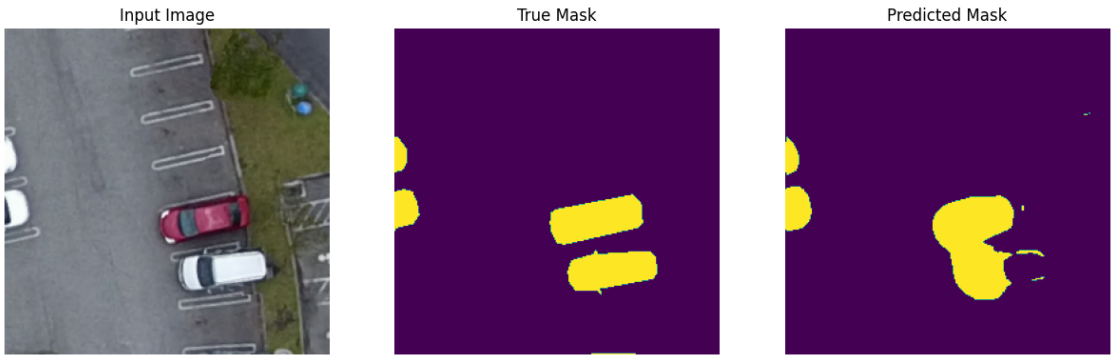

```
1/1 [==============================] - 0s 48ms/step
```



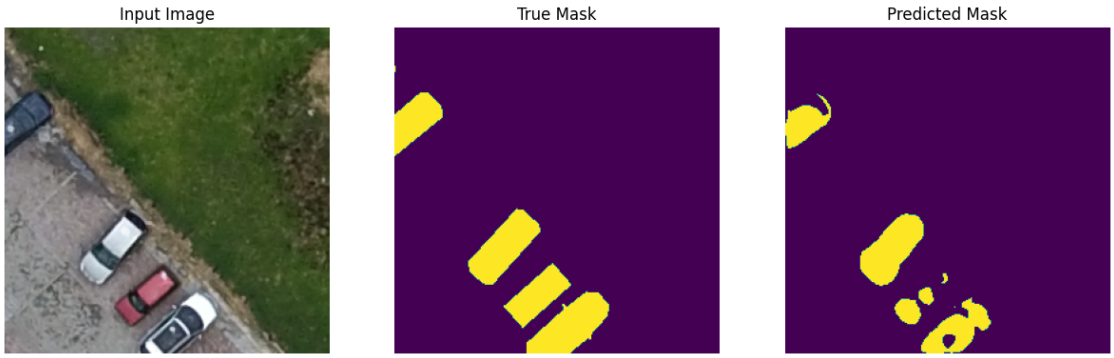

```
1/1 [==============================] - 0s 61ms/step
```



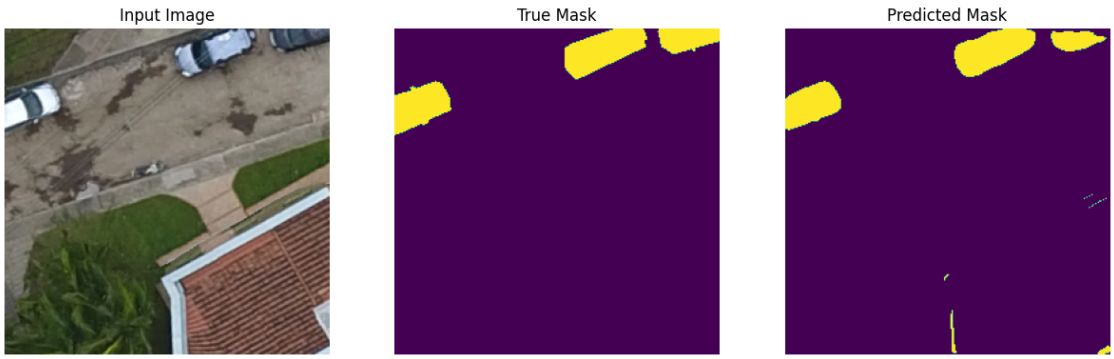

```
1/1 [==============================] - 0s 80ms/step
```



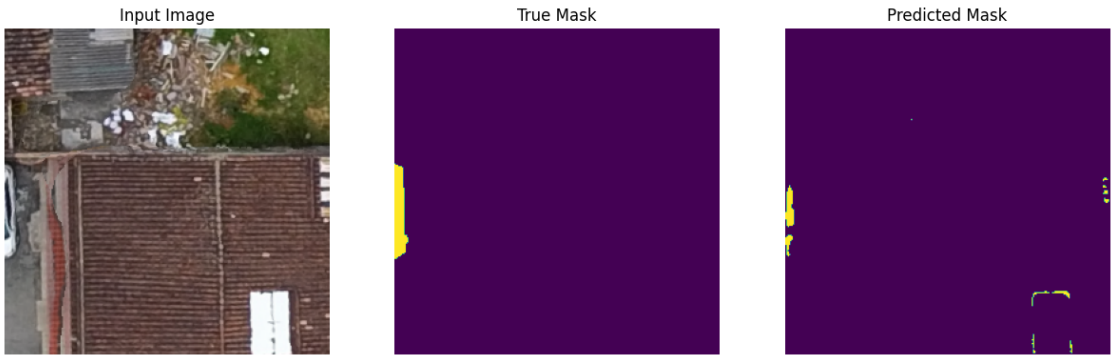

```
1/1 [==============================] - 0s 95ms/step
```




```
1/1 [==============================] - 0s 45ms/step
```

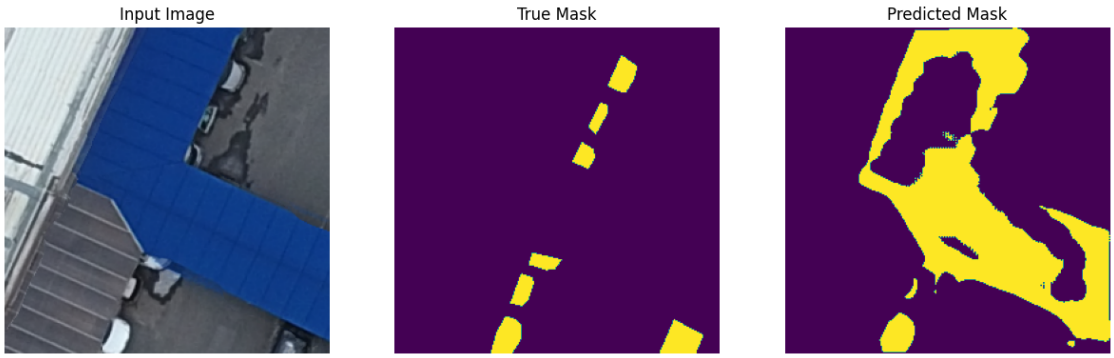



```
1/1 [==============================] - 0s 53ms/step
```



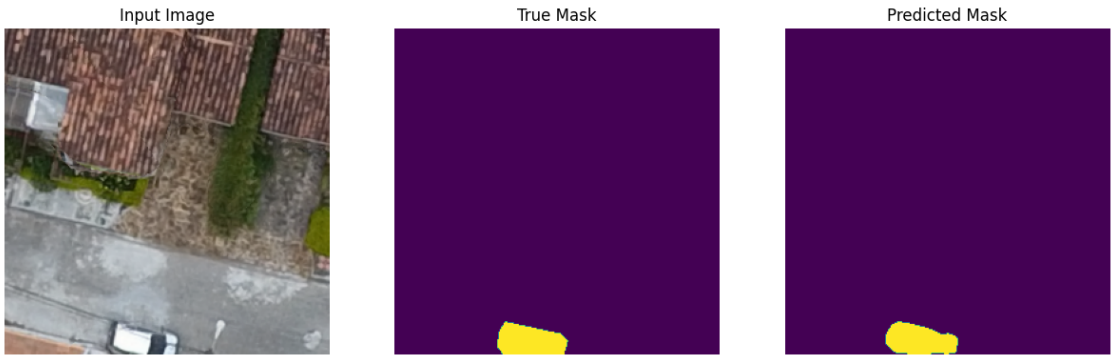

```
[ ]: show_predictions(test_batches, model_loaded, 16, 0.30)
```

```
1/1 [==============================] - 0s 39ms/step
```

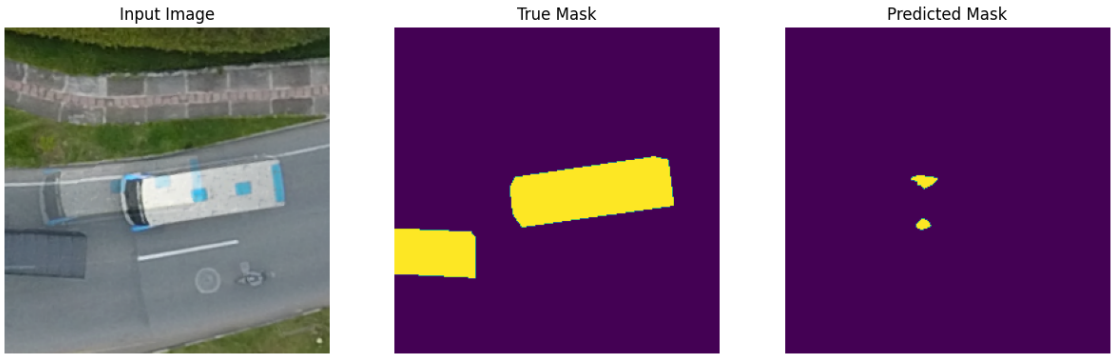



```
1/1 [==============================] - 0s 35ms/step
```

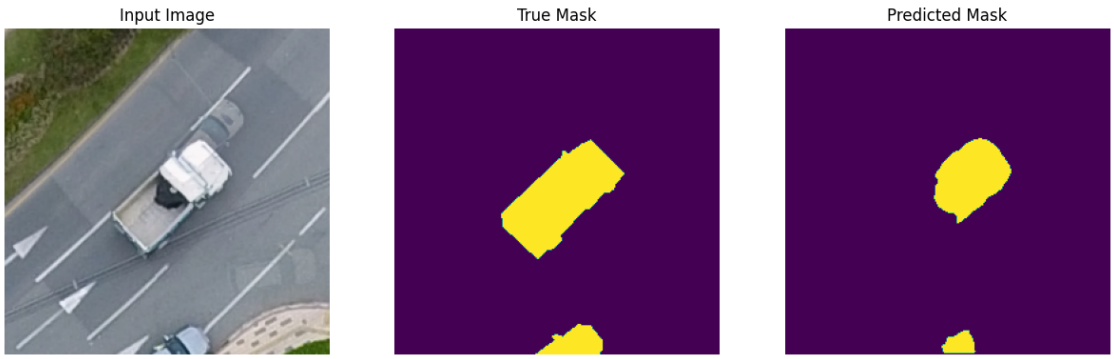

```
1/1 [==============================] - 0s 32ms/step
```



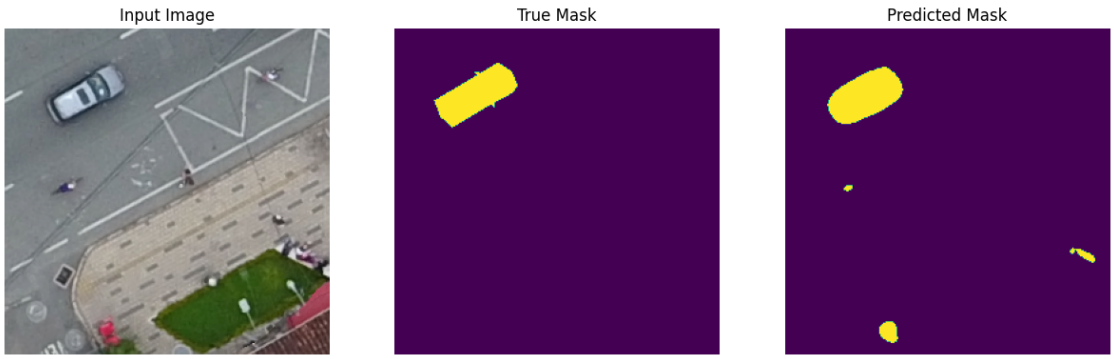

```
1/1 [==============================] - 0s 86ms/step
```



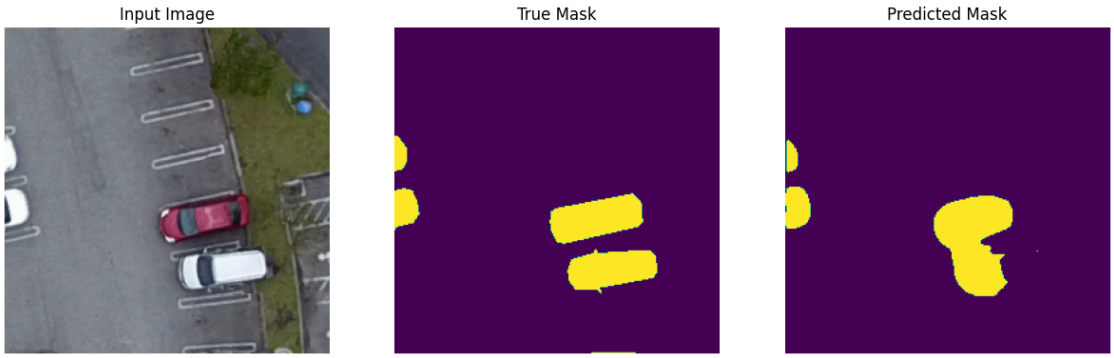

```
1/1 [==============================] - 0s 43ms/step
```



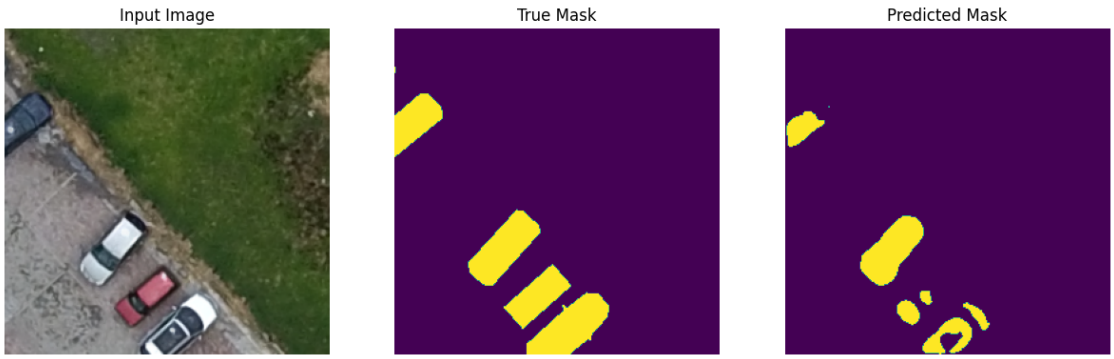

```
1/1 [==============================] - 0s 39ms/step
```



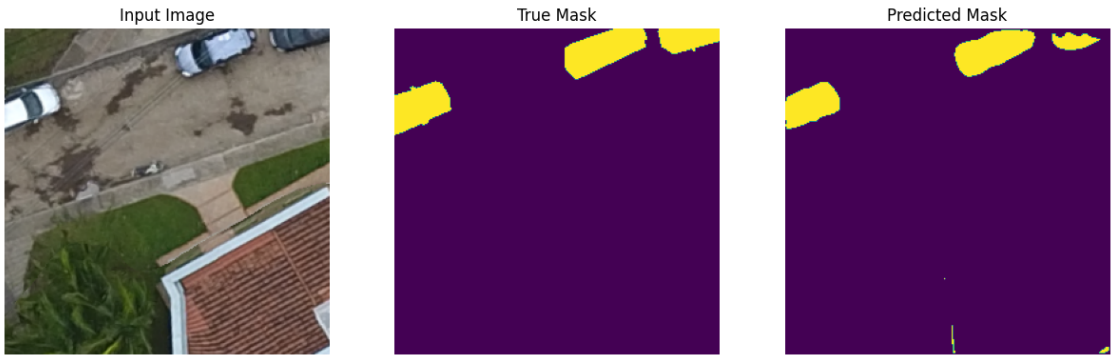

```
1/1 [==============================] - 0s 32ms/step
```



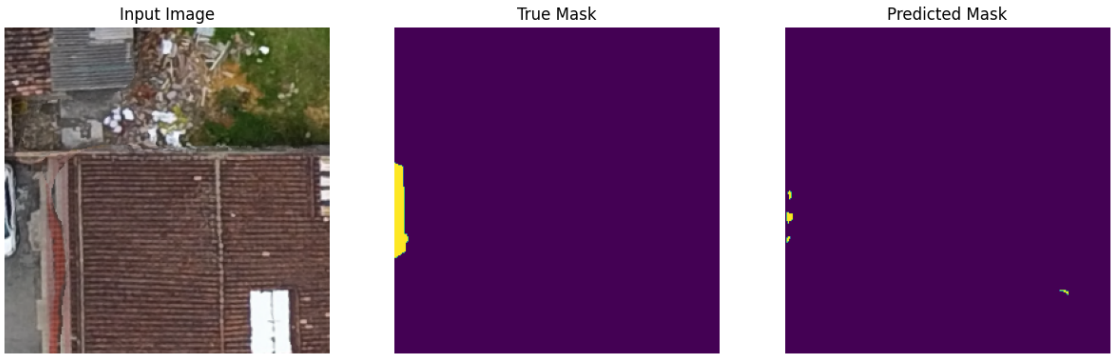

```
1/1 [==============================] - 0s 87ms/step
```




```
1/1 [==============================] - 0s 70ms/step
```

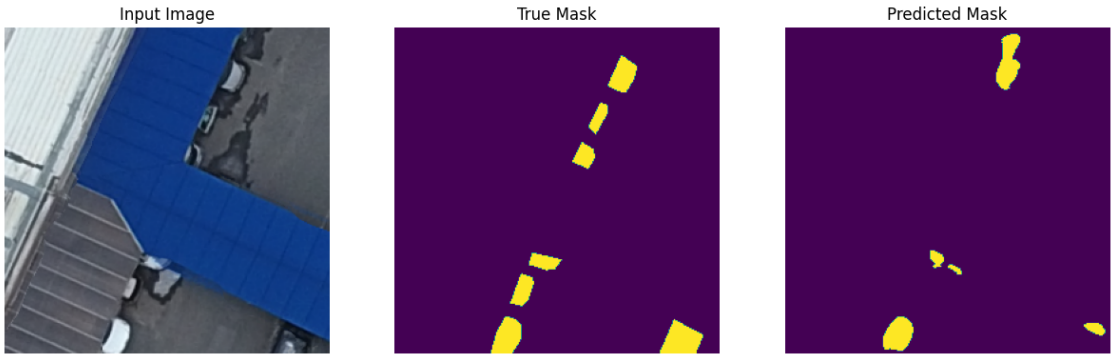



```
1/1 [==============================] - 0s 49ms/step
```



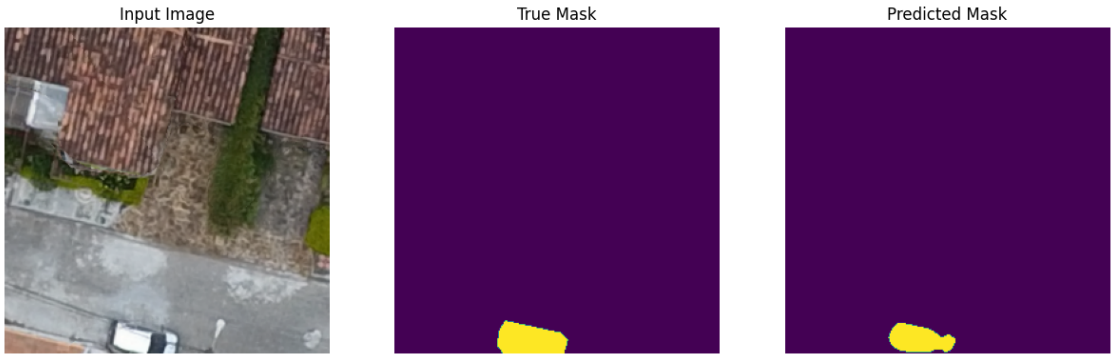
[ ]:
```
show_predictions(test_batches, model_loaded, 16, 0.40)
```

```
1/1 [==============================] - 0s 81ms/step
```

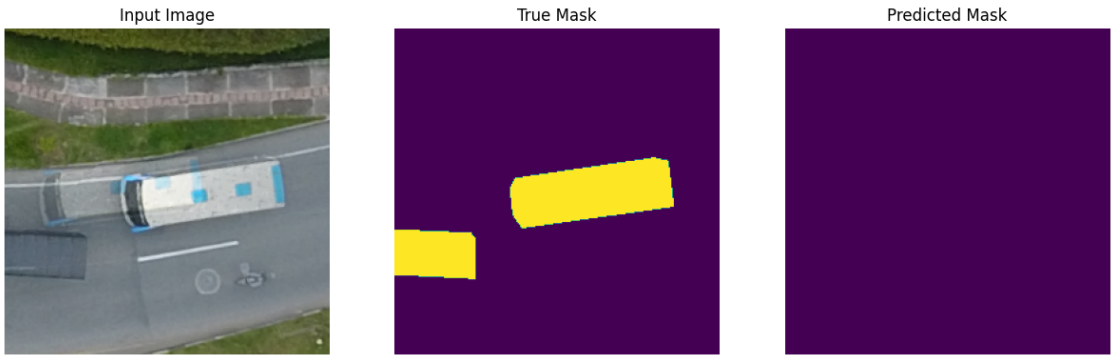



```
1/1 [==============================] - 0s 112ms/step
```

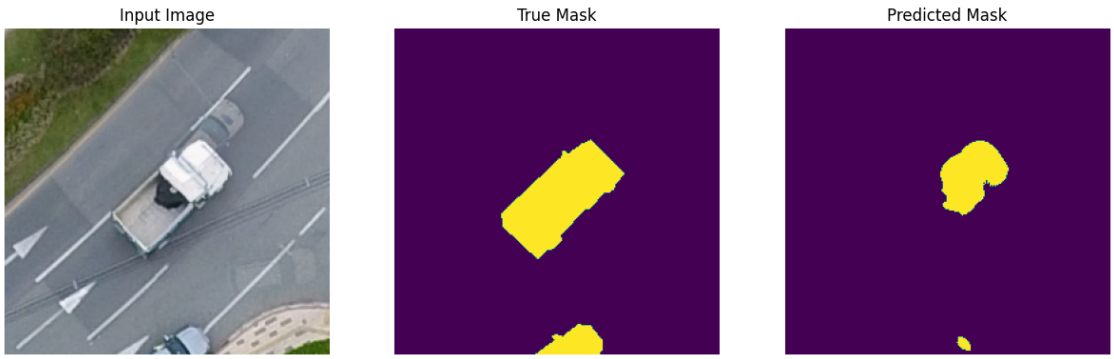

```
1/1 [==============================] - 0s 53ms/step
```



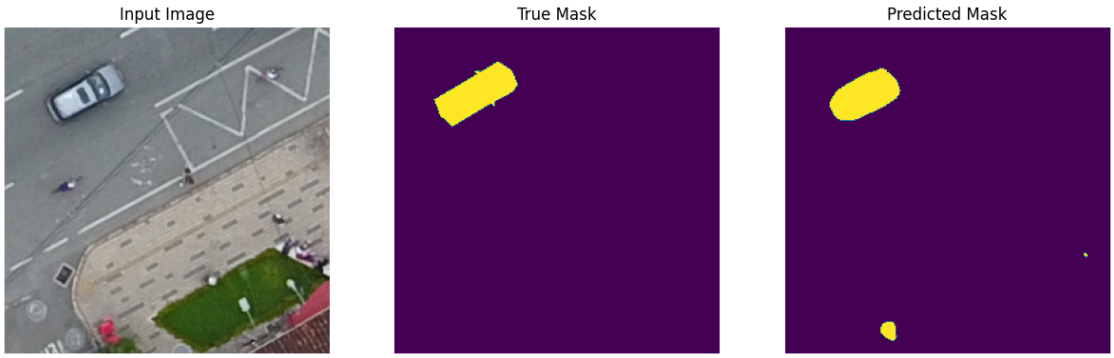

```
1/1 [==============================] - 0s 60ms/step
```



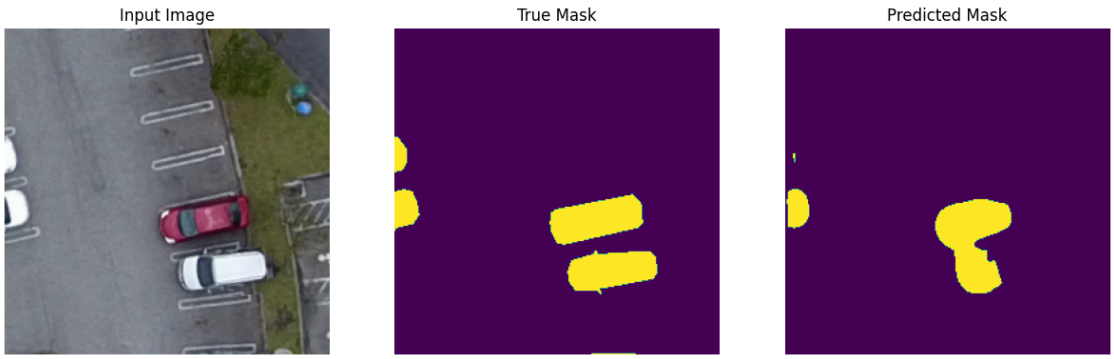

```
1/1 [==============================] - 0s 54ms/step
```



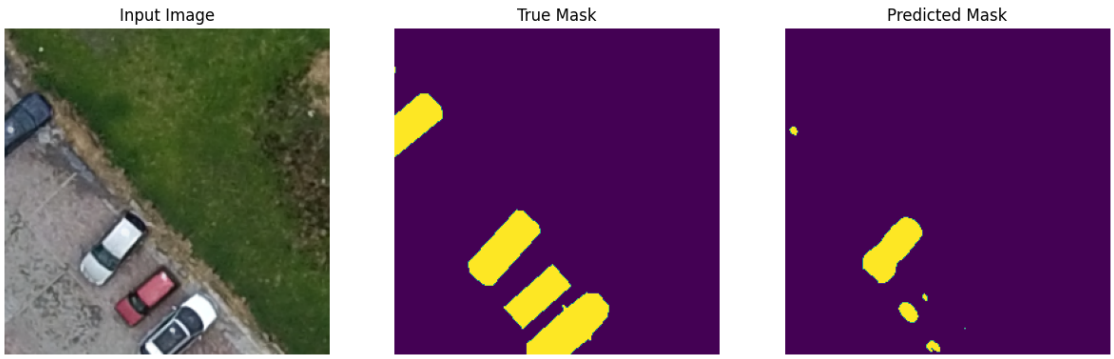

```
1/1 [==============================] - 0s 61ms/step
```



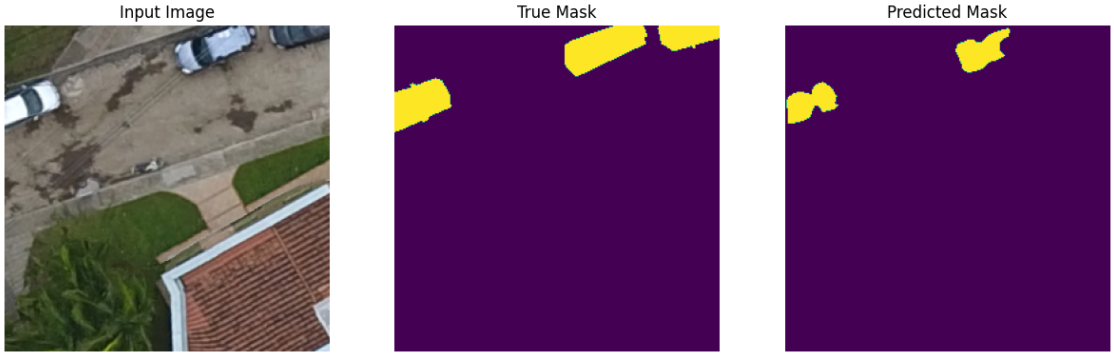

```
1/1 [==============================] - 0s 55ms/step
```



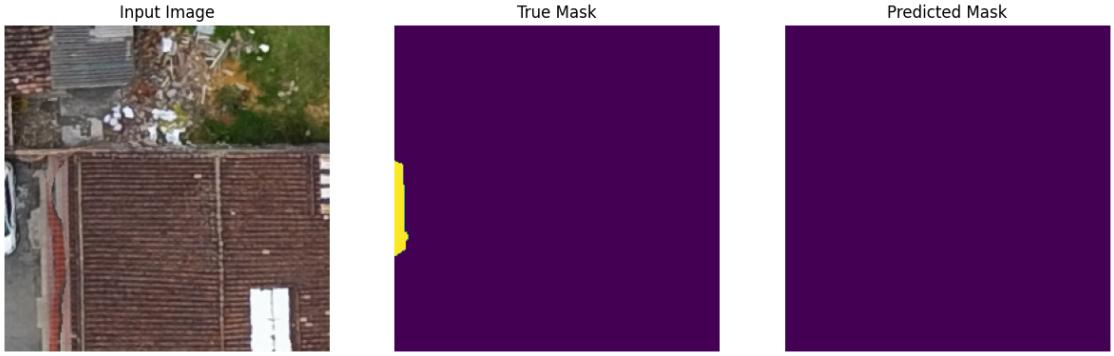

```
1/1 [==============================] - 0s 33ms/step
```




```
1/1 [==============================] - 0s 38ms/step
```

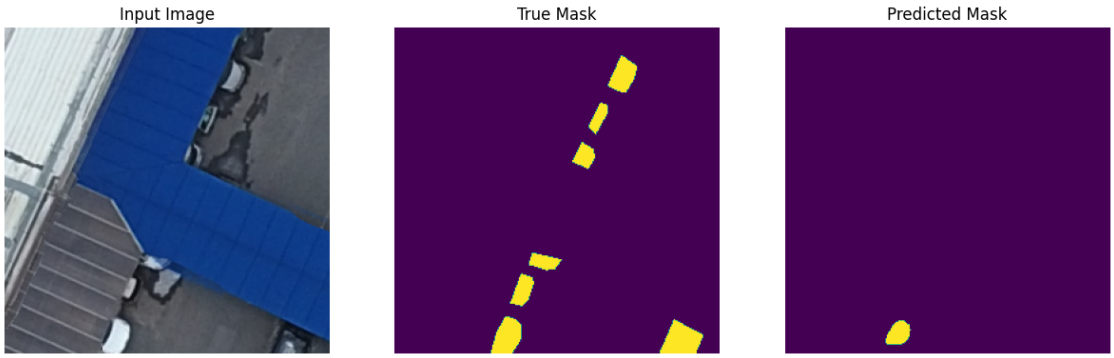



```
1/1 [==============================] - 0s 30ms/step
```



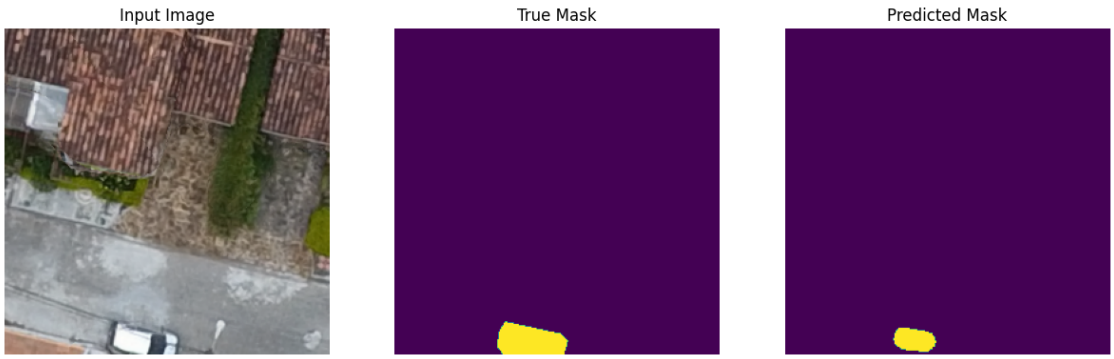
[ ]:
```
show_predictions(test_batches, model_loaded, 16, 0.50)
```

```
1/1 [==============================] - 0s 43ms/step
```

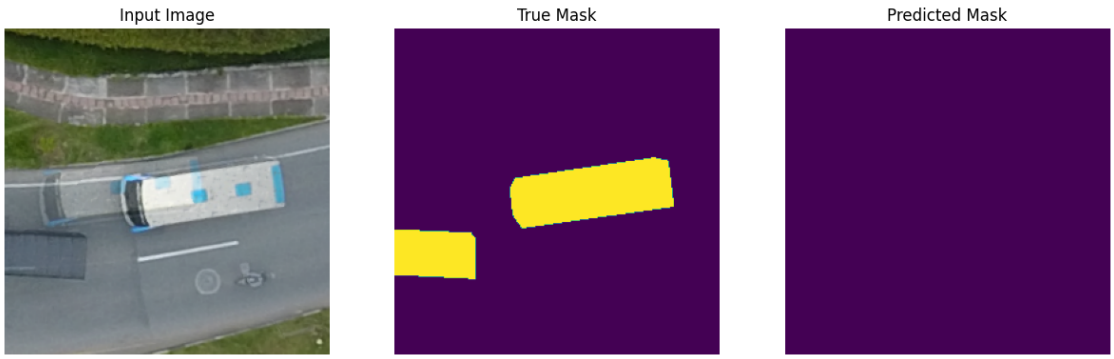



```
1/1 [==============================] - 0s 38ms/step
```

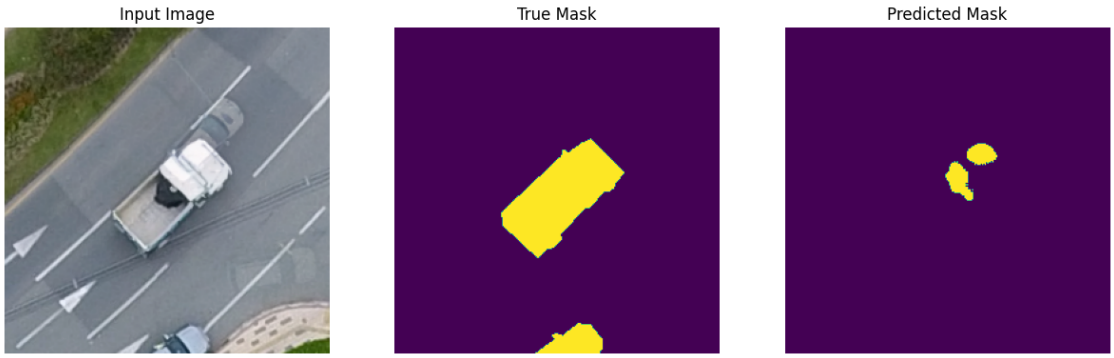

```
1/1 [==============================] - 0s 70ms/step
```



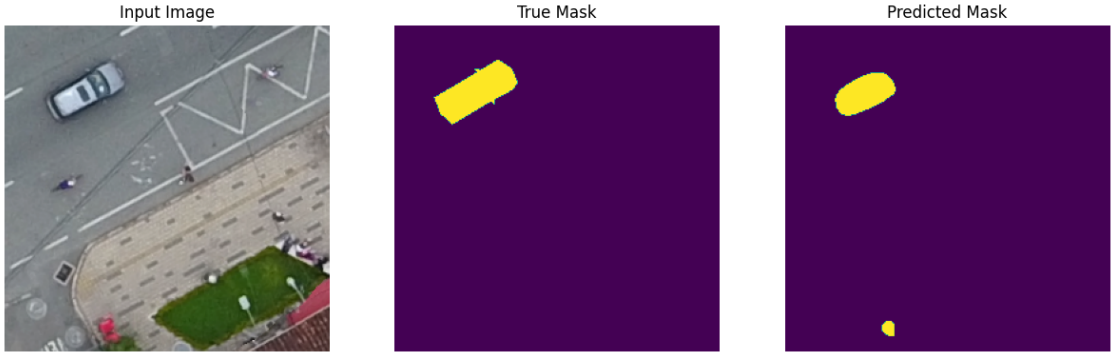

```
1/1 [==============================] - 0s 38ms/step
```



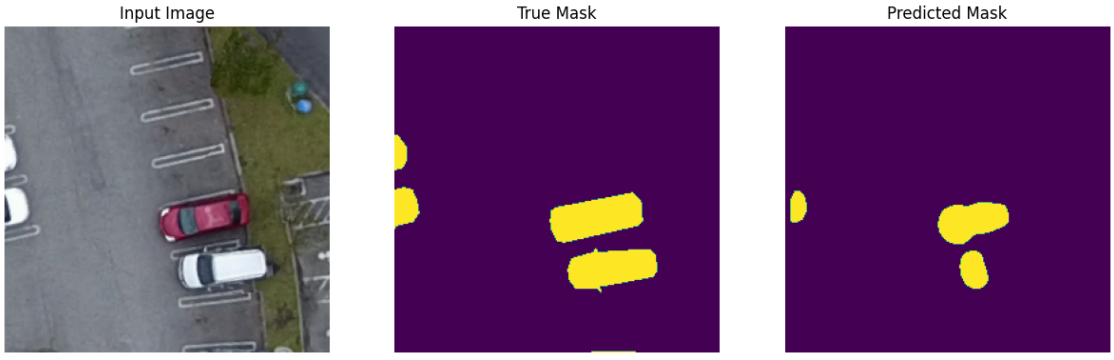

```
1/1 [==============================] - 0s 38ms/step
```



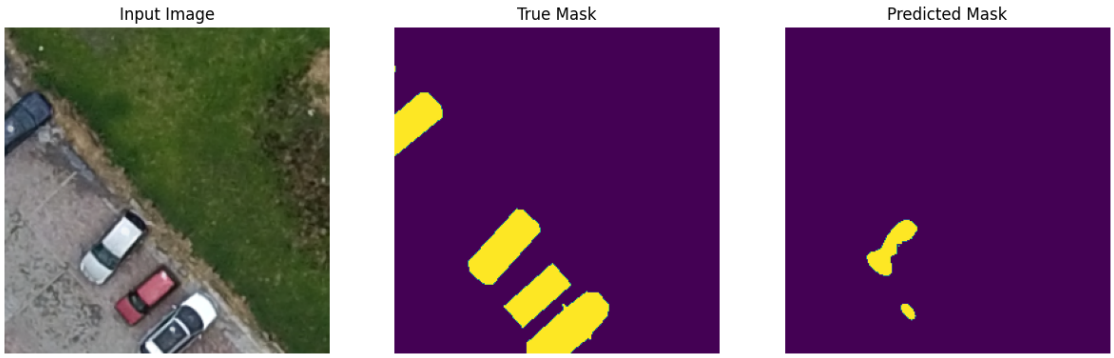

```
1/1 [==============================] - 0s 38ms/step
```



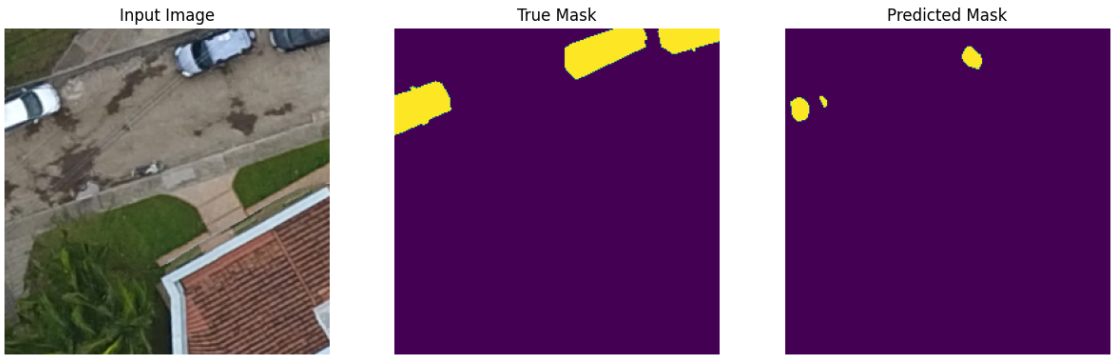

```
1/1 [==============================] - 0s 36ms/step
```




```
1/1 [==============================] - 0s 72ms/step
```




```
1/1 [==============================] - 0s 65ms/step
```




```
1/1 [==============================] - 0s 44ms/step
```