Notebook

March 3, 2024

```
[1]: from google.colab import drive
     drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

1 Image Segmentation Experiential Learning

2 Introduction

Image segmentation is a critical process in computer vision that involves partitioning an image into multiple segments or regions, each corresponding to distinct objects or parts of objects. This technique plays a pivotal role in various applications, including medical imaging, autonomous vehicles, object recognition, and augmented reality. In this project, we will focus on segmenting cat images to accurately identify and separate cats from their backgrounds. This project could potentially contribute to improving pet detection systems, cat breed classification, or even animal behavior analysis.

3 Literature Review:

3.1 Common methods used for Image segmentation

- Thresholding: Setting a fixed threshold value to divide the image into binary regions based on pixel intensity or color.
- Region-based segmentation: Grouping pixels with similar characteristics into regions using techniques like region growing or region merging.
- Edge-based segmentation: Detecting edges or boundaries in the image and separating different objects based on these edges.
- Clustering: Using clustering algorithms like k-means or mean-shift to group pixels with similar features into segments.
- Watershed segmentation: Treating the image as a topographic landscape and flooding it from markers to create distinct regions.
- Deep learning-based segmentation: Utilizing convolutional neural networks (CNNs) and deep learning techniques to learn complex representations for segmentation tasks. Popular architectures for this include U-Net, SegNet, and DeepLab.
- Markov Random Fields (MRFs) and Conditional Random Fields (CRFs): MRFs and CRFs are probabilistic graphical models used in image segmentation to model the spatial relationships

between pixels. They help incorporate contextual information and smoothness constraints into the segmentation process.

3.2 Why image segmentation in computer vision?

Image segmentation is essential for several reasons:

- Semantic understanding: Segmentation provides a more detailed and structured understanding of the content in an image. By labeling each region with a specific class or category, computer vision systems can gain a better grasp of the scene's semantics and context.
- Object recognition and detection: Image segmentation enables the identification and localization of objects within an image. Once an image is divided into segments, individual objects can be extracted and analyzed separately, making it easier to recognize and detect objects in complex scenes.
- Instance segmentation: In addition to classifying objects, image segmentation can also differentiate between multiple instances of the same object. This level of granularity is crucial in scenarios where there are multiple objects of the same type in an image, such as counting or tracking objects.
- Object tracking: Segmentation helps in tracking objects across frames in videos. By consistently segmenting the objects in each frame, their trajectories and movements can be analyzed over time.
- Scene understanding: For tasks like autonomous driving, scene understanding is crucial. Image segmentation can assist in identifying road boundaries, lane markings, pedestrians, and other vehicles, enabling the development of safer and more reliable autonomous systems.
- Image editing and manipulation: Segmentation allows the modification of specific regions within an image selectively. For example, it can be used to remove unwanted objects, change the background, or apply specific filters or effects only to certain regions.
- Medical imaging: In medical applications, image segmentation is used for various purposes, such as tumor detection, organ segmentation, and cell analysis, aiding in disease diagnosis and treatment planning.
- Image compression: Segmentation can help optimize image compression techniques by focusing more on preserving the important segments while reducing the complexity of less critical regions.

4 Dataset Exploration:

4.1 About Dataset

The Oxford-IIIT Pet Dataset is a 37 category pet dataset with roughly 200 images for each class created by the Visual Geometry Group at Oxford. The images have a large variations in scale, pose and lighting. All images have an associated ground truth annotation of breed, head ROI, and pixel level trimap segmentation.

4.1.1 Annotation Examples

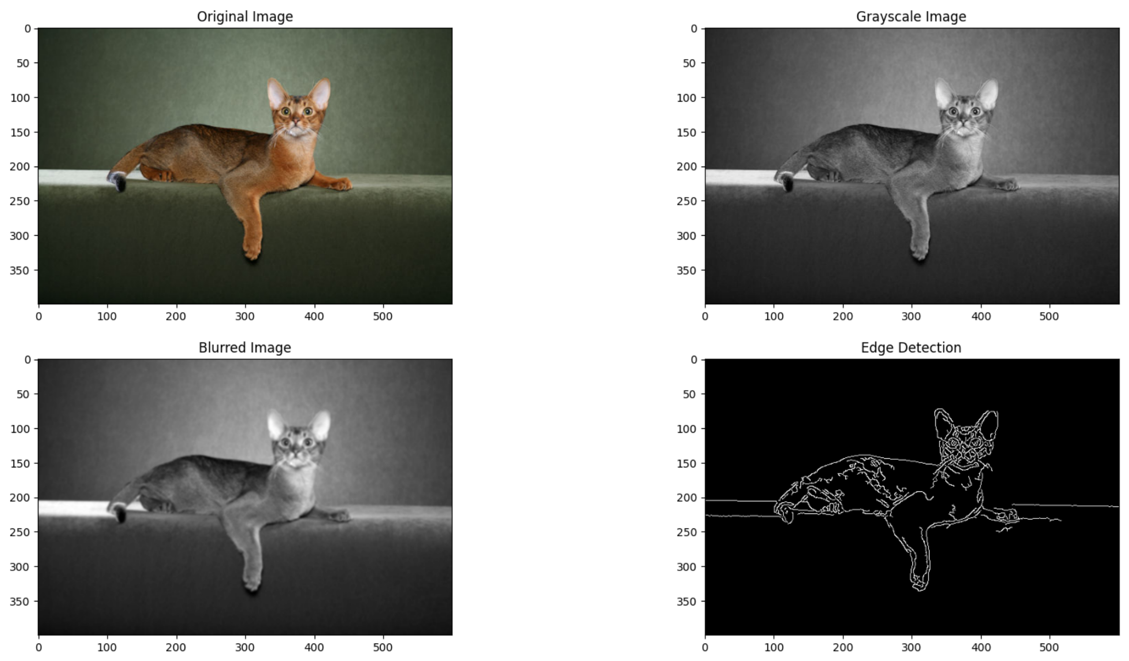
The following annotations are available for every image in the dataset: (a) species and breed name; (b) a tight bounding box (ROI) around the head of the animal; and (c) a pixel level foregroundbackground segmentation (Trimap).

Citation O. M. Parkhi et al., 2012

5 Hands-on Basic Image Processing:

[2]:
```
image_path = r"/content/drive/Othercomputers/astRo/SIT/TY - S2/CV/ImageSeg/
↪Abyssinian_1.jpg"
```

[3]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline


image = cv2.imread(image_path)

# Convert the image to grayscale
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Apply Gaussian blur
blurred_image = cv2.GaussianBlur(gray_image, (5, 5), 0)

# Perform edge detection
edges = cv2.Canny(blurred_image, 30, 100)

# Display the images
plt.figure(figsize=(20, 10))

plt.subplot(221), plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB)), plt.
↪title('Original Image')
plt.subplot(222), plt.imshow(gray_image, cmap='gray'), plt.title('Grayscale␣
↪Image')
plt.subplot(223), plt.imshow(blurred_image, cmap='gray'), plt.title('Blurred␣
↪Image')
plt.subplot(224), plt.imshow(edges, cmap='gray'), plt.title('Edge Detection')

plt.show()
```



6 Thresholding Techniques:

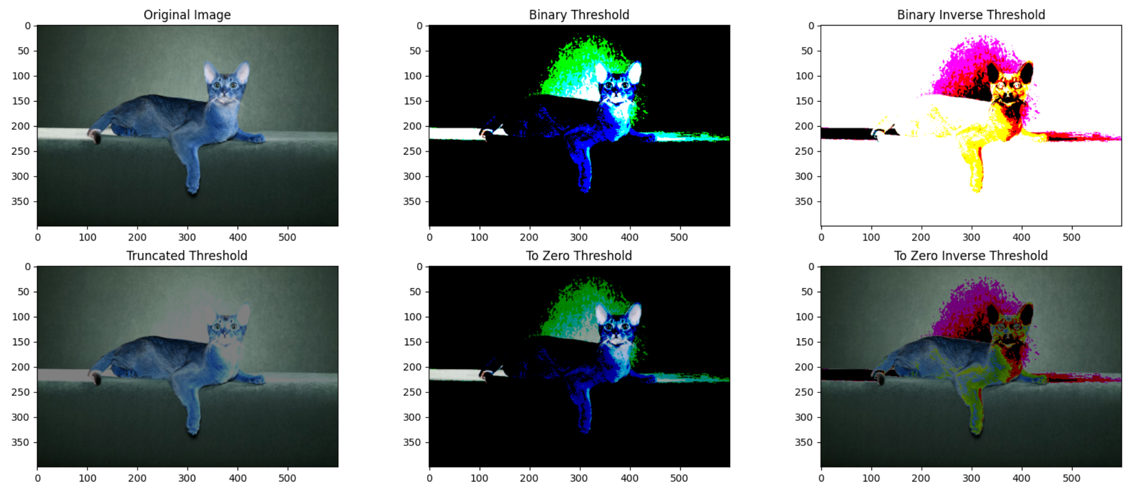
6.1 Different Thresholding Techniques

- Binary Thresholding
- Binary Inverse Thresholding
- Truncated Thresholding
- To Zero Thresholding
- To Zero Inverse Thresholding

- Otsu's Thresholding
- Adaptive Thresholding

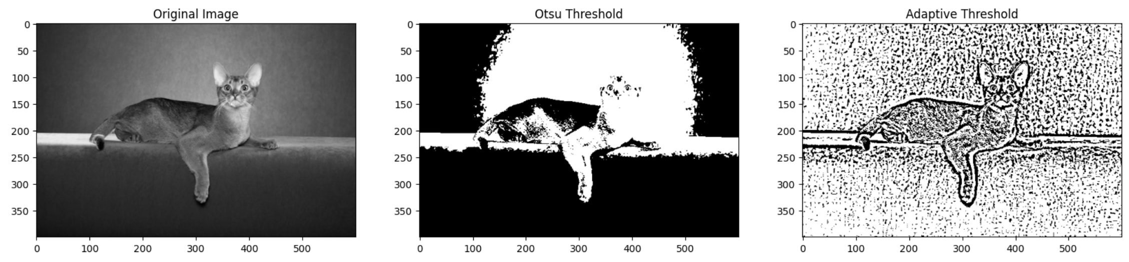
[4]:
```
# Apply different thresholding techniques
_, binary_thresh = cv2.threshold(image, 127, 255, cv2.THRESH_BINARY)
_, binary_inv_thresh = cv2.threshold(image, 127, 255, cv2.THRESH_BINARY_INV)
_, trunc_thresh = cv2.threshold(image, 127, 255, cv2.THRESH_TRUNC)
_, tozero_thresh = cv2.threshold(image, 127, 255, cv2.THRESH_TOZERO)
_, tozero_inv_thresh = cv2.threshold(image, 127, 255, cv2.THRESH_TOZERO_INV)

# Display the results
plt.figure(figsize=(20, 8))

plt.subplot(231), plt.imshow(image, cmap='gray'), plt.title('Original Image')
plt.subplot(232), plt.imshow(binary_thresh, cmap='gray'), plt.title('Binary␣
  ↪Threshold')
```

```
plt.subplot(233), plt.imshow(binary_inv_thresh, cmap='gray'), plt.title('Binary␣
  ↪Inverse Threshold')
plt.subplot(234), plt.imshow(trunc_thresh, cmap='gray'), plt.title('Truncated␣
  ↪Threshold')
plt.subplot(235), plt.imshow(tozero_thresh, cmap='gray'), plt.title('To Zero␣
  ↪Threshold')
plt.subplot(236), plt.imshow(tozero_inv_thresh, cmap='gray'), plt.title('To␣
  ↪Zero Inverse Threshold')

plt.show()
```



[5]:
```
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

# Apply Otsu's thresholding
_, otsu_thresh = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY + cv2.
  ↪THRESH_OTSU)

# Apply Adaptive thresholding
adaptive_thresh = cv2.adaptiveThreshold(image, 255, cv2.ADAPTIVE_THRESH_MEAN_C,␣
  ↪cv2.THRESH_BINARY, 11, 2)

# Display the results
plt.figure(figsize=(20, 20))

plt.subplot(131), plt.imshow(image, cmap='gray'), plt.title('Original Image')
plt.subplot(132), plt.imshow(otsu_thresh, cmap='gray'), plt.title('Otsu␣
  ↪Threshold')
plt.subplot(133), plt.imshow(adaptive_thresh, cmap='gray'), plt.title('Adaptive␣
  ↪Threshold')
```

```
plt.show()
```

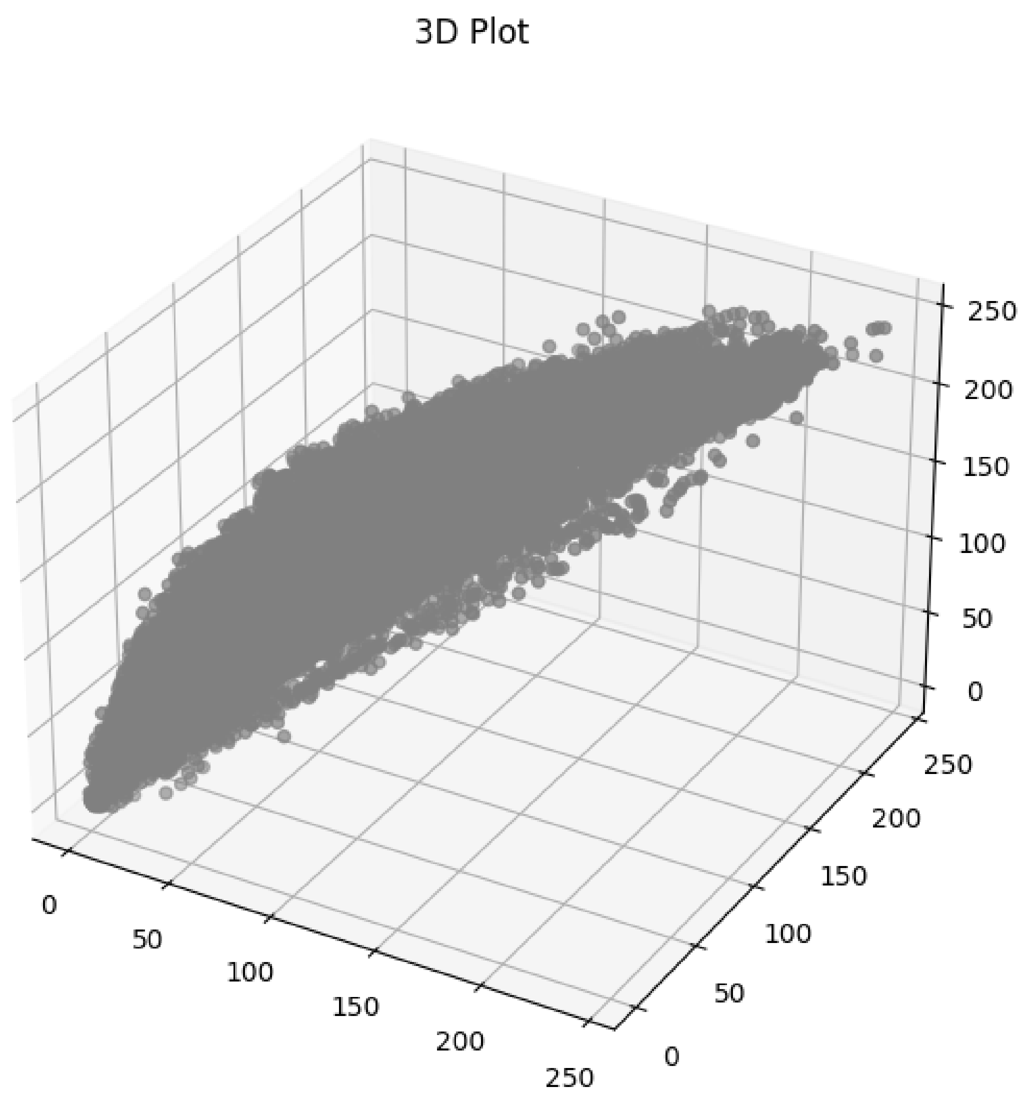


7 Clustering Algorithms:

- Apply clustering algorithms (e.g., K-means, DBSCAN) to segment images from your dataset. Evaluate the effectiveness of each algorithm.

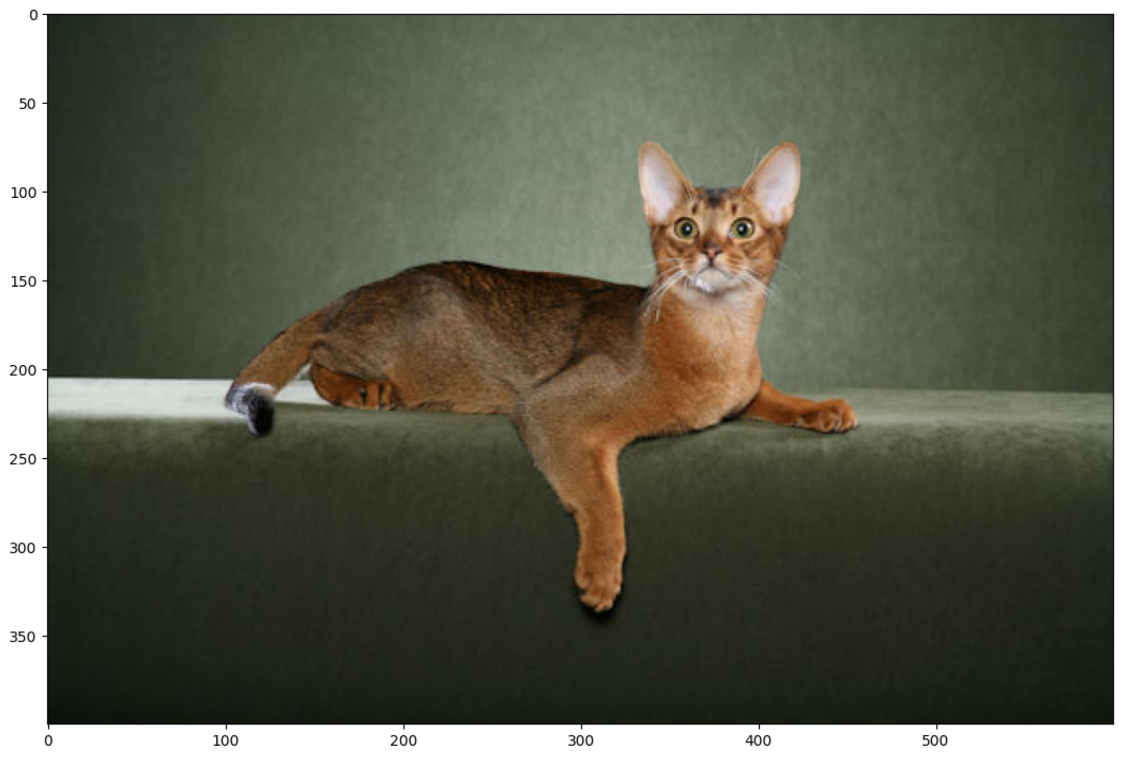
[ ]:
```
!pip install ipympl
```

[7]:
```
from mpl_toolkits.mplot3d import Axes3D

image = cv2.imread(image_path)
r, g, b = cv2.split(image)
r = r.flatten()
g = g.flatten()
b = b.flatten()#plotting

fig = plt.figure(figsize=(10, 7))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(r, g, b, c='0.5')
ax.set_title('3D Plot')
plt.show()
```



[8]:
```
img = cv2.imread(image_path)
img=cv2.cvtColor(img ,cv2.COLOR_BGR2RGB)
plt.figure(figsize=(13,10))
plt.imshow(img)
```

[8]:
```
<matplotlib.image.AxesImage at 0x7b6299a35570>
```



[9]:
```
img.shape #the first is height, the second is width and the third is the color␣
    ↪channel of the image
```

[9]:
```
(400, 600, 3)
```

[10]:
```
#Next, converts the HxWx3 image into a Kx3 matrix where K=HxW and each row is␣
    ↪now a vector in the 3-D space of RGB.
vectorized_img = img.reshape((-1,3))
vectorized_img.shape
```

[10]:
```
(240000, 3)
```

[11]:
```
import numpy as np
vectorized_img= np.float32(vectorized_img)
vectorized_img
```

[11]:
```
array([[30., 38., 27.],
       [30., 38., 27.],
       [30., 38., 27.],
       …,
       [15., 23., 12.],
       [15., 23., 12.],
       [15., 23., 12.]], dtype=float32)
```

[12]:
```
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
```

[13]:
```
K = 3
attempts=10
ret,label,center=cv2.kmeans(vectorized_img,K,None,criteria,attempts,cv2.
  ↪KMEANS_PP_CENTERS)
```

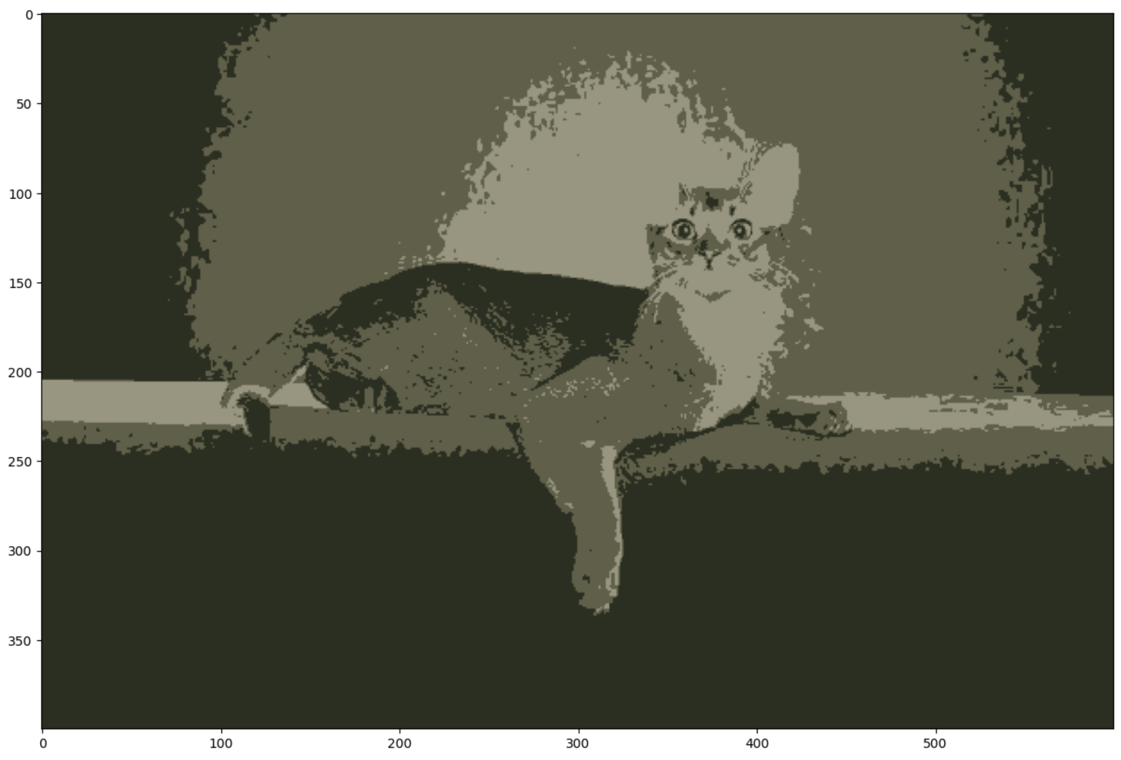
[14]:
```
center = np.uint8(center)
center
```

[14]:
```
array([[152, 150, 129],
       [ 43,  47,  33],
       [ 96,  96,  74]], dtype=uint8)
```

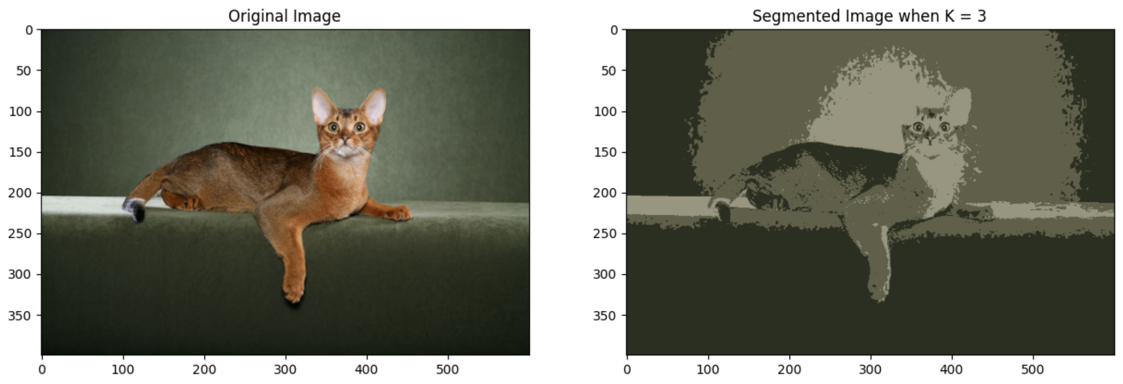
[15]:
```
#Next, we have to access the labels to regenerate the clustered image
res = center[label.flatten()]
result_image = res.reshape((img.shape))
```

[16]:
```
plt.figure(figsize=(15,10))
plt.imshow(result_image)
```

[16]:
```
<matplotlib.image.AxesImage at 0x7b6299af7e50>
```



[17]:
```
plt.figure(figsize=(15,12))
plt.subplot(1,2,1)
plt.imshow(img)
plt.title('Original Image')
plt.subplot(1,2,2)
plt.imshow(result_image)
plt.title('Segmented Image when K = %i' % K)
plt.show()
```



8 Deep Learning Model:

- Choose a pre-trained deep learning model (e.g., U-Net, DeepLab) and apply it to your dataset. Share the model, code, and results. Discuss the strengths and limitations of deep learning in image segmentation.

[18]:
```
!pip install git+https://github.com/tensorflow/examples.git
```

```
Collecting git+https://github.com/tensorflow/examples.git
  Cloning https://github.com/tensorflow/examples.git to /tmp/pip-req-
build-7j982qrl
  Running command git clone --filter=blob:none --quiet
https://github.com/tensorflow/examples.git /tmp/pip-req-build-7j982qrl
  Resolved https://github.com/tensorflow/examples.git to commit
fff4bcda7201645a1efaea4534403daf5fc03d42
  Preparing metadata (setup.py) … done
Requirement already satisfied: absl-py in /usr/local/lib/python3.10/dist-
packages (from tensorflow-
examples==0.1703207612.146125047983137092961436282825516886814646O245314)
(1.4.0)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages
(from tensorflow-
examples==0.1703207612.1461250479831370929614362828255168868146460245314)
(1.16.0)
```

[19]:
```
import tensorflow as tf

import tensorflow_datasets as tfds
```

[20]:
```
from tensorflow_examples.models.pix2pix import pix2pix

from IPython.display import clear_output
import matplotlib.pyplot as plt
```

8.1 Load the Oxford-IIIT Pets dataset

[21]:
```
dataset, info = tfds.load('oxford_iiit_pet:3.*.*', with_info=True)
```

[22]:
```
def normalize(input_image, input_mask):
  input_image = tf.cast(input_image, tf.float32) / 255.0
  input_mask -= 1
  return input_image, input_mask

def load_image(datapoint):
  input_image = tf.image.resize(datapoint['image'], (128, 128))
  input_mask = tf.image.resize(
    datapoint['segmentation_mask'],
    (128, 128),
    method = tf.image.ResizeMethod.NEAREST_NEIGHBOR,
  )

  input_image, input_mask = normalize(input_image, input_mask)

  return input_image, input_mask
```

[23]:
```
TRAIN_LENGTH = info.splits['train'].num_examples
BATCH_SIZE = 64
BUFFER_SIZE = 1000
STEPS_PER_EPOCH = TRAIN_LENGTH // BATCH_SIZE

train_images = dataset['train'].map(load_image, num_parallel_calls=tf.data.
  ↪AUTOTUNE)
test_images = dataset['test'].map(load_image, num_parallel_calls=tf.data.
  ↪AUTOTUNE)
```

The following class performs a simple augmentation by randomly-flipping an image.

[24]:
```
class Augment(tf.keras.layers.Layer):
  def __init__(self, seed=42):
    super().__init__()
    # both use the same seed, so they'll make the same random changes.
    self.augment_inputs = tf.keras.layers.RandomFlip(mode="horizontal",␣
  ↪seed=seed)
```

```
    self.augment_labels = tf.keras.layers.RandomFlip(mode="horizontal",␣
↪seed=seed)

  def call(self, inputs, labels):
    inputs = self.augment_inputs(inputs)
    labels = self.augment_labels(labels)
    return inputs, labels
```

Build the input pipeline, applying the augmentation after batching the inputs:

[25]:
```
train_batches = (
    train_images
    .cache()
    .shuffle(BUFFER_SIZE)
    .batch(BATCH_SIZE)
    .repeat()
    .map(Augment())
    .prefetch(buffer_size=tf.data.AUTOTUNE))

test_batches = test_images.batch(BATCH_SIZE)
```

Visualize an image example and its corresponding mask from the dataset

[26]:
```
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()
```

[27]:
```
for images, masks in train_batches.take(2):
  sample_image, sample_mask = images[0], masks[0]
  display([sample_image, sample_mask])
```

8.2 Define the model

The model being used here is a modified U-Net. A U-Net consists of an encoder (downsampler) and decoder (upsampler). To learn robust features and reduce the number of trainable parameters, use a pretrained model—MobileNetV2—as the encoder. For the decoder, we will use the upsample block, which is already implemented in the pix2pix example in the TensorFlow Examples repo.

As mentioned, the encoder is a pretrained MobileNetV2 model. You will use the model from `tf.keras.applications`. The encoder consists of specific outputs from intermediate layers in the model. Note that the encoder will not be trained during the training process.

[28]:
```
base_model = tf.keras.applications.MobileNetV2(input_shape=[128, 128, 3],␣
 ↪include_top=False)

# Use the activations of these layers
layer_names = [
    'block_1_expand_relu',   # 64x64
    'block_3_expand_relu',   # 32x32
    'block_6_expand_relu',   # 16x16
    'block_13_expand_relu',  # 8x8
    'block_16_project',      # 4x4
]
base_model_outputs = [base_model.get_layer(name).output for name in layer_names]

# Create the feature extraction model
down_stack = tf.keras.Model(inputs=base_model.input, outputs=base_model_outputs)

down_stack.trainable = False
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applicatio
ns/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_128_no_top.h
5
9406464/9406464 [==============================] - 1s 0us/step
```

The decoder/upsampler is simply a series of upsample block.

[29]:
```
up_stack = [
    pix2pix.upsample(512, 3),  # 4x4 -> 8x8
    pix2pix.upsample(256, 3),  # 8x8 -> 16x16
    pix2pix.upsample(128, 3),  # 16x16 -> 32x32
    pix2pix.upsample(64, 3),   # 32x32 -> 64x64
]
```

[30]:
```
def unet_model(output_channels:int):
  inputs = tf.keras.layers.Input(shape=[128, 128, 3])

  # Downsampling through the model
  skips = down_stack(inputs)
  x = skips[-1]
  skips = reversed(skips[:-1])

  # Upsampling and establishing the skip connections
  for up, skip in zip(up_stack, skips):
    x = up(x)
    concat = tf.keras.layers.Concatenate()
    x = concat([x, skip])

  # This is the last layer of the model
  last = tf.keras.layers.Conv2DTranspose(
      filters=output_channels, kernel_size=3, strides=2,
      padding='same')  #64x64 -> 128x128

  x = last(x)

  return tf.keras.Model(inputs=inputs, outputs=x)
```

Note that the number of filters on the last layer is set to the number of `output_channels`. This will be one output channel per class.

8.3 Train the model

Now, all that is left to do is to compile and train the model.

Since this is a multiclass classification problem, use the `tf.keras.losses.SparseCategoricalCrossentropy` loss function with the `from_logits` argument set to `True`, since the labels are scalar integers instead of vectors of scores for each pixel of every class.

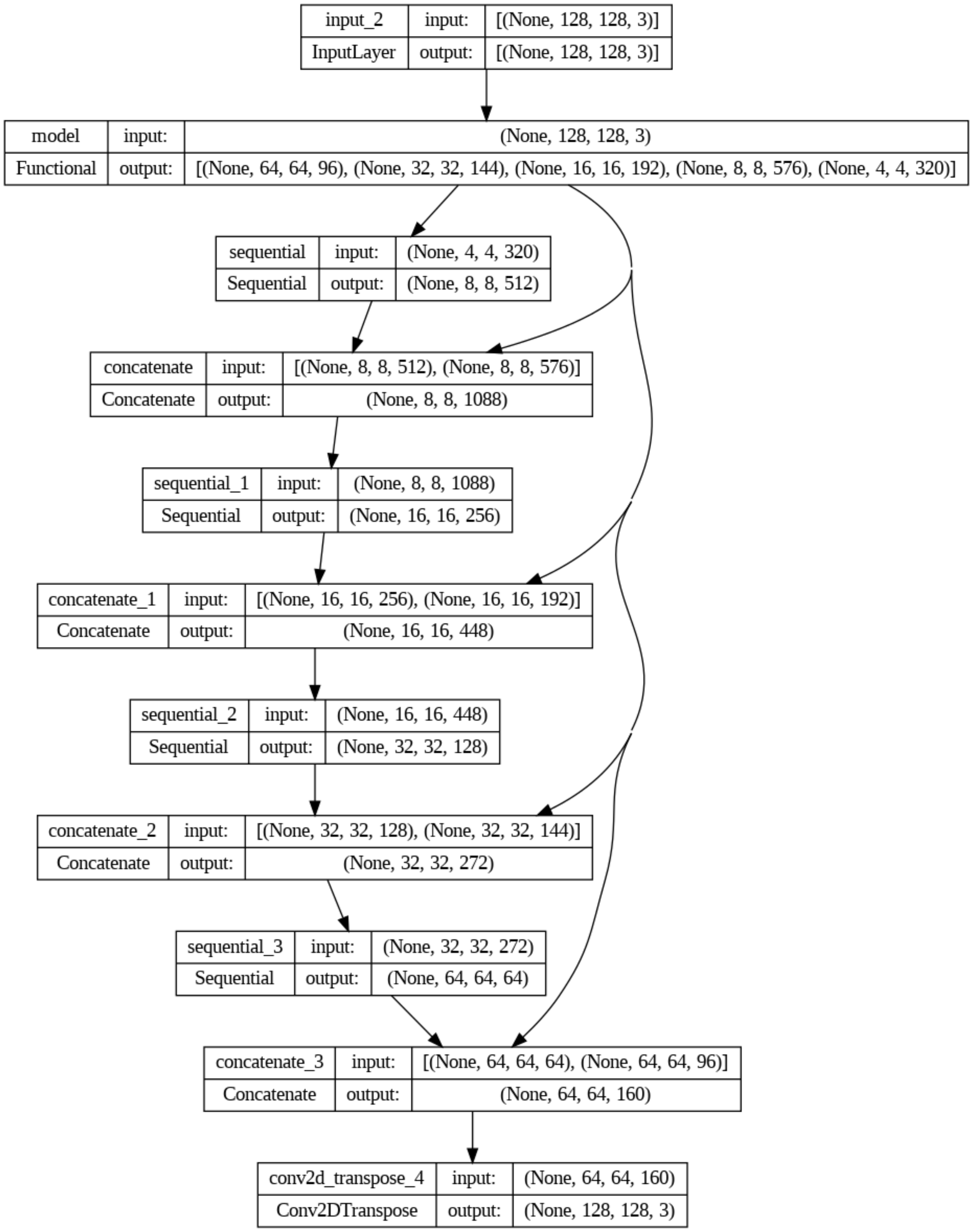
When running inference, the label assigned to the pixel is the channel with the highest value. This is what the `create_mask` function is doing.

[31]:
```
OUTPUT_CLASSES = 3

model = unet_model(output_channels=OUTPUT_CLASSES)
model.compile(optimizer='adam',
              loss=tf.keras.losses.
 ↪SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Plot the resulting model architecture:

[32]:
```
tf.keras.utils.plot_model(model, show_shapes=True)
```

[32]:



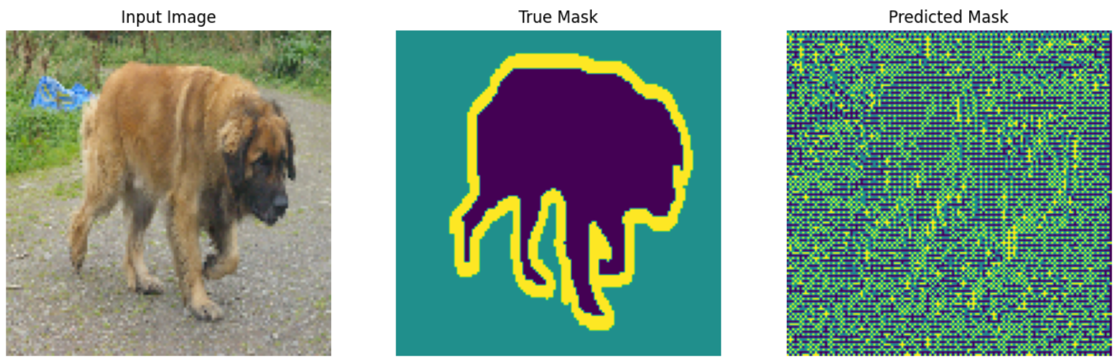
input_2 | input: | [(None, 128, 128, 3)]
InputLayer | output: | [(None, 128, 128, 3)]
model | input: | (None, 128, 128, 3)
Functional | output: | [(None, 64, 64, 96), (None, 32, 32, 144), (None, 16, 16, 192), (None, 8, 8, 576), (None, 4, 4, 320)]
sequential | input: | (None, 4, 4, 320)
Sequential | output: | (None, 8, 8, 512)
concatenate | input: | [(None, 8, 8, 512), (None, 8, 8, 576)]
Concatenate | output: | (None, 8, 8, 1088)
sequential_1 | input: | (None, 8, 8, 1088)
Sequential | output: | (None, 16, 16, 256)
concatenate_1 | input: | [(None, 16, 16, 256), (None, 16, 16, 192)]
Concatenate | output: | (None, 16, 16, 448)
sequential_2 | input: | (None, 16, 16, 448)
Sequential | output: | (None, 32, 32, 128)
concatenate_2 | input: | [(None, 32, 32, 128), (None, 32, 32, 144)]
Concatenate | output: | (None, 32, 32, 272)
sequential_3 | input: | (None, 32, 32, 272)
Sequential | output: | (None, 64, 64, 64)
concatenate_3 | input: | [(None, 64, 64, 64), (None, 64, 64, 96)]
Concatenate | output: | (None, 64, 64, 160)
conv2d_transpose_4 | input: | (None, 64, 64, 160)
Conv2DTranspose | output: | (None, 128, 128, 3)

[33]:
```
def create_mask(pred_mask):
  pred_mask = tf.math.argmax(pred_mask, axis=-1)
  pred_mask = pred_mask[..., tf.newaxis]
  return pred_mask[0]
```

[34]:
```
def show_predictions(dataset=None, num=1):
  if dataset:
    for image, mask in dataset.take(num):
      pred_mask = model.predict(image)
      display([image[0], mask[0], create_mask(pred_mask)])
  else:
    display([sample_image, sample_mask,
             create_mask(model.predict(sample_image[tf.newaxis, ...]))])
```

[35]:
```
show_predictions()
```

```
1/1 [==============================] - 5s 5s/step
```



The callback defined below is used to observe how the model improves while it is training:

[36]:
```
class DisplayCallback(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs=None):
    clear_output(wait=True)
    show_predictions()
    print ('\nSample Prediction after epoch {}\n'.format(epoch+1))
```

[37]:
```
import tensorflow as tf
tf.test.is_built_with_cuda()
tf.test.is_gpu_available(cuda_only=False, min_cuda_compute_capability=None)
```

```
WARNING:tensorflow:From <ipython-input-37-09365002d536>:3: is_gpu_available
(from tensorflow.python.framework.test_util) is deprecated and will be removed
in a future version.
Instructions for updating:
Use `tf.config.list_physical_devices('GPU')` instead.
```

[37]:
```
True
```

[42]:
```
EPOCHS = 50
VAL_SUBSPLITS = 5
```

```python
VALIDATION_STEPS = info.splits['test'].num_examples//BATCH_SIZE//VAL_SUBSPLITS

model_history = model.fit(train_batches, epochs=EPOCHS,
                          steps_per_epoch=STEPS_PER_EPOCH,
                          validation_steps=VALIDATION_STEPS,
                          validation_data=test_batches)
```

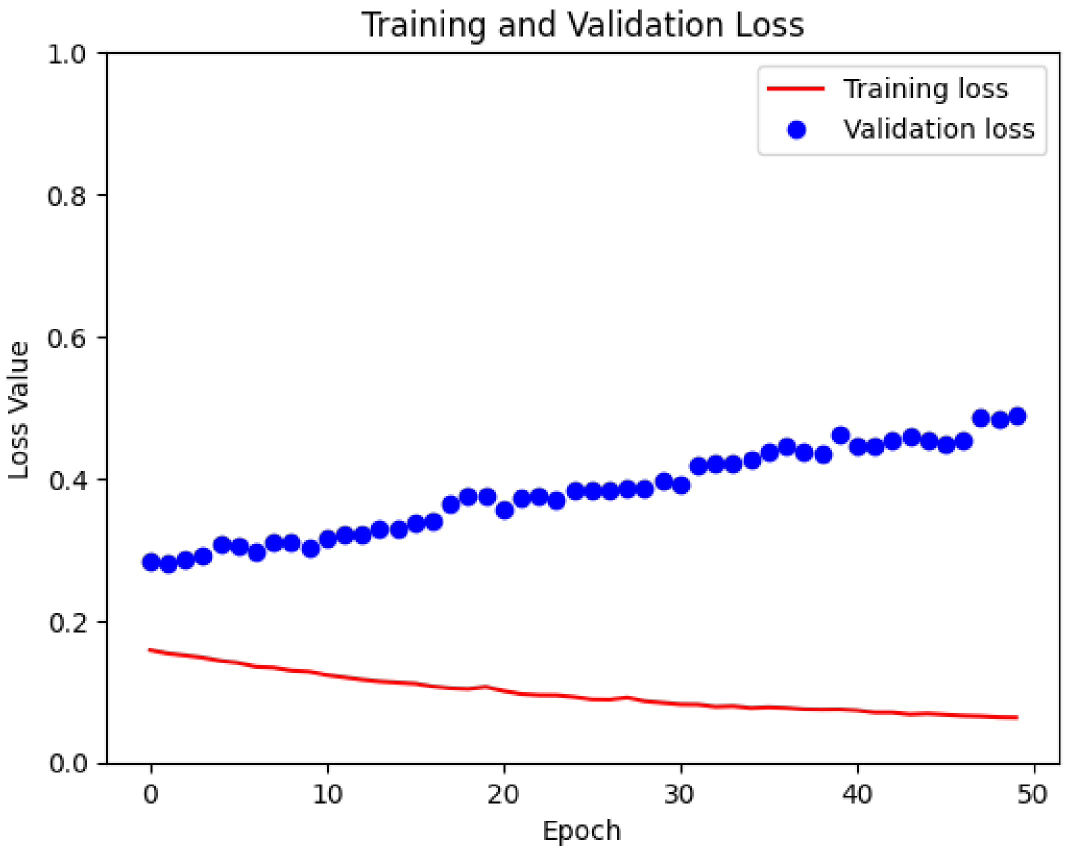
```
Epoch 1/50
57/57 [==============================] - 9s 163ms/step - loss: 0.1584 -
accuracy: 0.9350 - val_loss: 0.2840 - val_accuracy: 0.9055
Epoch 2/50
57/57 [==============================] - 9s 151ms/step - loss: 0.1537 -
accuracy: 0.9367 - val_loss: 0.2817 - val_accuracy: 0.9052
Epoch 3/50
57/57 [==============================] - 12s 210ms/step - loss: 0.1510 -
accuracy: 0.9378 - val_loss: 0.2868 - val_accuracy: 0.9045
Epoch 4/50
57/57 [==============================] - 9s 162ms/step - loss: 0.1479 -
accuracy: 0.9390 - val_loss: 0.2904 - val_accuracy: 0.9017
Epoch 5/50
57/57 [==============================] - 9s 151ms/step - loss: 0.1433 -
accuracy: 0.9408 - val_loss: 0.3071 - val_accuracy: 0.9029
Epoch 6/50
57/57 [==============================] - 9s 165ms/step - loss: 0.1405 -
accuracy: 0.9419 - val_loss: 0.3055 - val_accuracy: 0.9027
Epoch 7/50
57/57 [==============================] - 12s 207ms/step - loss: 0.1349 -
accuracy: 0.9441 - val_loss: 0.2976 - val_accuracy: 0.9037
Epoch 8/50
57/57 [==============================] - 9s 161ms/step - loss: 0.1337 -
accuracy: 0.9445 - val_loss: 0.3101 - val_accuracy: 0.9037
Epoch 9/50
57/57 [==============================] - 9s 162ms/step - loss: 0.1296 -
accuracy: 0.9462 - val_loss: 0.3111 - val_accuracy: 0.9039
Epoch 10/50
57/57 [==============================] - 8s 148ms/step - loss: 0.1281 -
accuracy: 0.9468 - val_loss: 0.3029 - val_accuracy: 0.9030
Epoch 11/50
57/57 [==============================] - 9s 162ms/step - loss: 0.1233 -
accuracy: 0.9486 - val_loss: 0.3152 - val_accuracy: 0.9034
Epoch 12/50
57/57 [==============================] - 9s 163ms/step - loss: 0.1201 -
accuracy: 0.9500 - val_loss: 0.3213 - val_accuracy: 0.9023
Epoch 13/50
57/57 [==============================] - 9s 164ms/step - loss: 0.1169 -
accuracy: 0.9513 - val_loss: 0.3207 - val_accuracy: 0.9040
Epoch 14/50
57/57 [==============================] - 12s 207ms/step - loss: 0.1144 -
```

```
accuracy: 0.9523 - val_loss: 0.3290 - val_accuracy: 0.9029
Epoch 15/50
57/57 [==============================] - 8s 145ms/step - loss: 0.1128 -
accuracy: 0.9529 - val_loss: 0.3281 - val_accuracy: 0.9027
Epoch 16/50
57/57 [==============================] - 8s 148ms/step - loss: 0.1114 -
accuracy: 0.9536 - val_loss: 0.3364 - val_accuracy: 0.9030
Epoch 17/50
57/57 [==============================] - 9s 164ms/step - loss: 0.1072 -
accuracy: 0.9552 - val_loss: 0.3408 - val_accuracy: 0.9029
Epoch 18/50
57/57 [==============================] - 8s 146ms/step - loss: 0.1049 -
accuracy: 0.9561 - val_loss: 0.3635 - val_accuracy: 0.9007
Epoch 19/50
57/57 [==============================] - 8s 149ms/step - loss: 0.1038 -
accuracy: 0.9566 - val_loss: 0.3743 - val_accuracy: 0.9011
Epoch 20/50
57/57 [==============================] - 9s 163ms/step - loss: 0.1067 -
accuracy: 0.9554 - val_loss: 0.3766 - val_accuracy: 0.9015
Epoch 21/50
57/57 [==============================] - 8s 145ms/step - loss: 0.1009 -
accuracy: 0.9578 - val_loss: 0.3564 - val_accuracy: 0.9016
Epoch 22/50
57/57 [==============================] - 8s 148ms/step - loss: 0.0966 -
accuracy: 0.9596 - val_loss: 0.3716 - val_accuracy: 0.8985
Epoch 23/50
57/57 [==============================] - 10s 168ms/step - loss: 0.0951 -
accuracy: 0.9602 - val_loss: 0.3756 - val_accuracy: 0.9019
Epoch 24/50
57/57 [==============================] - 9s 161ms/step - loss: 0.0949 -
accuracy: 0.9603 - val_loss: 0.3706 - val_accuracy: 0.8991
Epoch 25/50
57/57 [==============================] - 9s 164ms/step - loss: 0.0924 -
accuracy: 0.9613 - val_loss: 0.3843 - val_accuracy: 0.9021
Epoch 26/50
57/57 [==============================] - 9s 163ms/step - loss: 0.0890 -
accuracy: 0.9627 - val_loss: 0.3836 - val_accuracy: 0.9025
Epoch 27/50
57/57 [==============================] - 8s 145ms/step - loss: 0.0887 -
accuracy: 0.9629 - val_loss: 0.3848 - val_accuracy: 0.9030
Epoch 28/50
57/57 [==============================] - 9s 165ms/step - loss: 0.0917 -
accuracy: 0.9617 - val_loss: 0.3861 - val_accuracy: 0.9025
Epoch 29/50
57/57 [==============================] - 9s 157ms/step - loss: 0.0865 -
accuracy: 0.9638 - val_loss: 0.3867 - val_accuracy: 0.9002
Epoch 30/50
57/57 [==============================] - 12s 210ms/step - loss: 0.0843 -
```

```
accuracy: 0.9647 - val_loss: 0.3973 - val_accuracy: 0.8996
Epoch 31/50
57/57 [==============================] - 9s 163ms/step - loss: 0.0821 -
accuracy: 0.9656 - val_loss: 0.3927 - val_accuracy: 0.9003
Epoch 32/50
57/57 [==============================] - 10s 172ms/step - loss: 0.0818 -
accuracy: 0.9657 - val_loss: 0.4196 - val_accuracy: 0.9011
Epoch 33/50
57/57 [==============================] - 9s 163ms/step - loss: 0.0788 -
accuracy: 0.9670 - val_loss: 0.4216 - val_accuracy: 0.9002
Epoch 34/50
57/57 [==============================] - 8s 146ms/step - loss: 0.0796 -
accuracy: 0.9666 - val_loss: 0.4222 - val_accuracy: 0.9007
Epoch 35/50
57/57 [==============================] - 8s 149ms/step - loss: 0.0770 -
accuracy: 0.9677 - val_loss: 0.4266 - val_accuracy: 0.8972
Epoch 36/50
57/57 [==============================] - 10s 168ms/step - loss: 0.0780 -
accuracy: 0.9673 - val_loss: 0.4380 - val_accuracy: 0.9002
Epoch 37/50
57/57 [==============================] - 9s 161ms/step - loss: 0.0771 -
accuracy: 0.9677 - val_loss: 0.4465 - val_accuracy: 0.9000
Epoch 38/50
57/57 [==============================] - 8s 149ms/step - loss: 0.0751 -
accuracy: 0.9686 - val_loss: 0.4375 - val_accuracy: 0.8990
Epoch 39/50
57/57 [==============================] - 12s 207ms/step - loss: 0.0747 -
accuracy: 0.9687 - val_loss: 0.4341 - val_accuracy: 0.9011
Epoch 40/50
57/57 [==============================] - 9s 161ms/step - loss: 0.0749 -
accuracy: 0.9686 - val_loss: 0.4621 - val_accuracy: 0.8978
Epoch 41/50
57/57 [==============================] - 9s 167ms/step - loss: 0.0736 -
accuracy: 0.9692 - val_loss: 0.4448 - val_accuracy: 0.8992
Epoch 42/50
57/57 [==============================] - 9s 150ms/step - loss: 0.0706 -
accuracy: 0.9704 - val_loss: 0.4460 - val_accuracy: 0.8988
Epoch 43/50
57/57 [==============================] - 10s 179ms/step - loss: 0.0705 -
accuracy: 0.9705 - val_loss: 0.4545 - val_accuracy: 0.8976
Epoch 44/50
57/57 [==============================] - 8s 146ms/step - loss: 0.0678 -
accuracy: 0.9716 - val_loss: 0.4581 - val_accuracy: 0.8992
Epoch 45/50
57/57 [==============================] - 8s 149ms/step - loss: 0.0688 -
accuracy: 0.9711 - val_loss: 0.4530 - val_accuracy: 0.8997
Epoch 46/50
57/57 [==============================] - 12s 208ms/step - loss: 0.0673 -
```

```
accuracy: 0.9718 - val_loss: 0.4485 - val_accuracy: 0.8981
Epoch 47/50
57/57 [==============================] - 8s 144ms/step - loss: 0.0659 -
accuracy: 0.9724 - val_loss: 0.4543 - val_accuracy: 0.9000
Epoch 48/50
57/57 [==============================] - 8s 149ms/step - loss: 0.0652 -
accuracy: 0.9726 - val_loss: 0.4856 - val_accuracy: 0.8997
Epoch 49/50
57/57 [==============================] - 9s 160ms/step - loss: 0.0639 -
accuracy: 0.9732 - val_loss: 0.4828 - val_accuracy: 0.8995
Epoch 50/50
57/57 [==============================] - 9s 161ms/step - loss: 0.0635 -
accuracy: 0.9734 - val_loss: 0.4891 - val_accuracy: 0.8995
```

[43]:
```
loss = model_history.history['loss']
val_loss = model_history.history['val_loss']

plt.figure()
plt.plot(model_history.epoch, loss, 'r', label='Training loss')
plt.plot(model_history.epoch, val_loss, 'bo', label='Validation loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss Value')
plt.ylim([0, 1])
plt.legend()
plt.show()
```

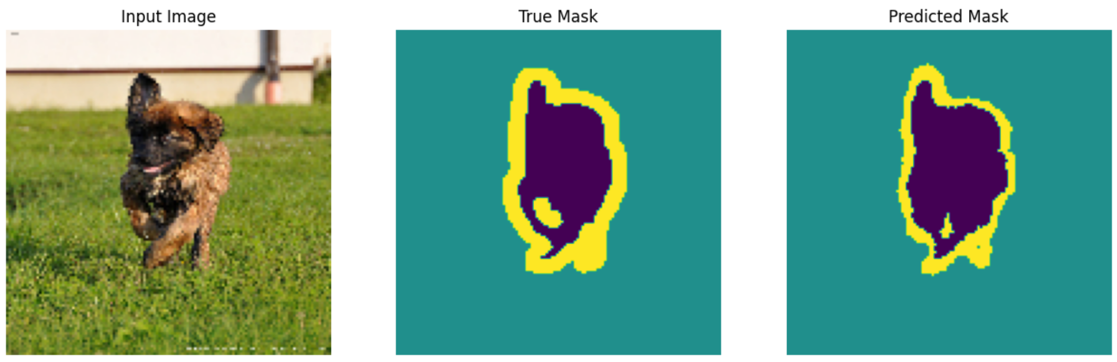


8.4 Make predictions

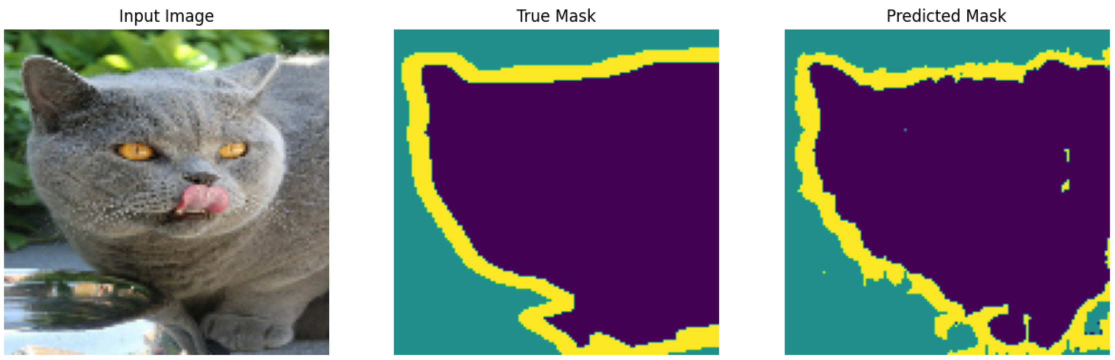
Now, make some predictions. In the interest of saving time, the number of epochs was kept small, but you may set this higher to achieve more accurate results.

```
[44]: show_predictions(test_batches, 3)
```

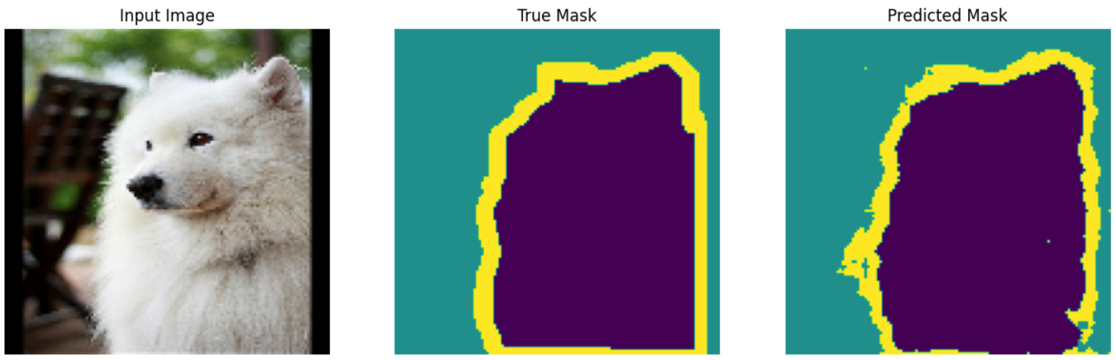
```
2/2 [==============================] - 0s 43ms/step
```



```
2/2 [==============================] - 0s 45ms/step
```



```
2/2 [==============================] - 0s 43ms/step
```



```
[ ]:
```