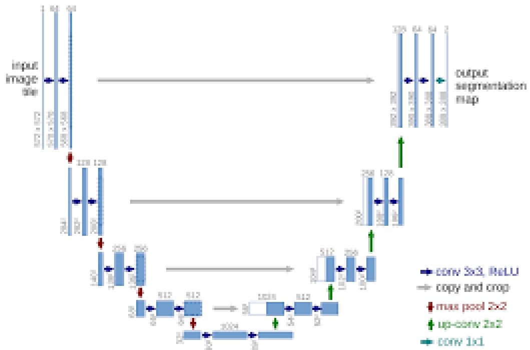
Image Segmentation using U-Net on Oxford-IIIT Pet Dataset

- This project implements image segmentation using the U-Net architecture on the Oxford-IIIT Pet Dataset. Image segmentation is a computer vision task that involves labeling each pixel in an image with its corresponding class, allowing for detailed understanding of objects and their boundaries. In our case, we classify each pixel as either pet, background, or outline.



- The U-Net model, known for its U-shaped structure, consists of a contracting path to capture context and an expanding path to enable precise localization using skip connections. The Oxford-IIIT Pet Dataset contains over 7,000 images of 37 pet breeds with corresponding segmentation masks. The model processes these images to generate pixel-level predictions, enabling accurate pet segmentation with well-defined edges and minimal background noise.

Importing libraries

In [2]:
```
import tensorflow as tf
import tensorflow.keras
import keras
import tensorflow_datasets as tfds
from tensorflow.keras import layers, Sequential , models
from keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Dropout
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
```

Loading the dataset

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

Preprocessing

To prepare the Oxford-IIIT Pet Dataset for training, several preprocessing steps were applied. First, each image and its corresponding segmentation mask were resized to a fixed dimension of 128x128 pixels. The pixel values of the images were normalized to the range [0, 1], and the mask values were adjusted by subtracting 1 to ensure class labels start from zero. For data augmentation, a random horizontal flip was applied to increase the diversity of training samples and improve generalization. The dataset was then batched, shuffled with a buffer size of 1000, and prefetching was enabled to ensure optimal performance during training. These preprocessing steps help stabilize and speed up the training process while improving model robustness.

In [4]:
```
len(dataset['train'])
```

Out[4]:
```
3680
```

In [5]:
```
info
```

Out[5]:
```
tfds.core.DatasetInfo(
    name='oxford_iiit_pet',
    full_name='oxford_iiit_pet/4.0.0',
    description="""
    The Oxford-IIIT pet dataset is a 37 category pet image dataset with rou
ghly 200
    images for each class. The images have large variations in scale, pose
and
    lighting. All images have an associated ground truth annotation of bree
d and
    species. Additionally, head bounding boxes are provided for the trainin
g split,
    allowing using this dataset for simple object detection tasks. In the t
est
    split, the bounding boxes are empty.
    """,
    homepage='http://www.robots.ox.ac.uk/~vgg/data/pets/',
    data_dir='/root/tensorflow_datasets/oxford_iiit_pet/4.0.0',
    file_format=tfrecord,
    download_size=773.52 MiB,
    dataset_size=773.68 MiB,
    features=FeaturesDict({
        'file_name': Text(shape=(), dtype=string),
        'head_bbox': BBoxFeature(shape=(4,), dtype=float32),
        'image': Image(shape=(None, None, 3), dtype=uint8),
        'label': ClassLabel(shape=(), dtype=int64, num_classes=37),
        'segmentation_mask': Image(shape=(None, None, 1), dtype=uint8),
        'species': ClassLabel(shape=(), dtype=int64, num_classes=2),
    }),
    supervised_keys=('image', 'label'),
    disable_shuffling=False,
    nondeterministic_order=False,
    splits={
        'test': <SplitInfo num_examples=3669, num_shards=4>,
        'train': <SplitInfo num_examples=3680, num_shards=4>,
    },
    citation="""@InProceedings{parkhi12a,
      author       = "Parkhi, O. M. and Vedaldi, A. and Zisserman, A. and J
awahar, C.~V.",
      title        = "Cats and Dogs",
      booktitle    = "IEEE Conference on Computer Vision and Pattern Recogn
ition",
      year         = "2012",
    }""",
)
```

In [6]:
```
def normalize(input_image, input_mask):
  input_image = tf.cast(input_image, tf.float32)/255
  input_mask = input_mask-1
  return input_image, input_mask
```

In [7]:
```
def load_train_images(sample):
  input_image= tf.image.resize(sample['image'],(128,128))
  input_mask = tf.image.resize(sample['segmentation_mask'],(128,128))
  if tf.random.uniform(())>0.5:
```

```
    input_image = tf.image.flip_left_right(input_image)
    input_mask = tf.image.flip_left_right(input_mask)
  input_image, input_mask = normalize(input_image, input_mask)
  return input_image , input_mask
```

In [8]:
```
def load_test_images(sample):
  input_image= tf.image.resize(sample['image'],(128,128))
  input_mask = tf.image.resize(sample['segmentation_mask'],(128,128))
  input_image, input_mask = normalize(input_image, input_mask)
  return input_image , input_mask
```

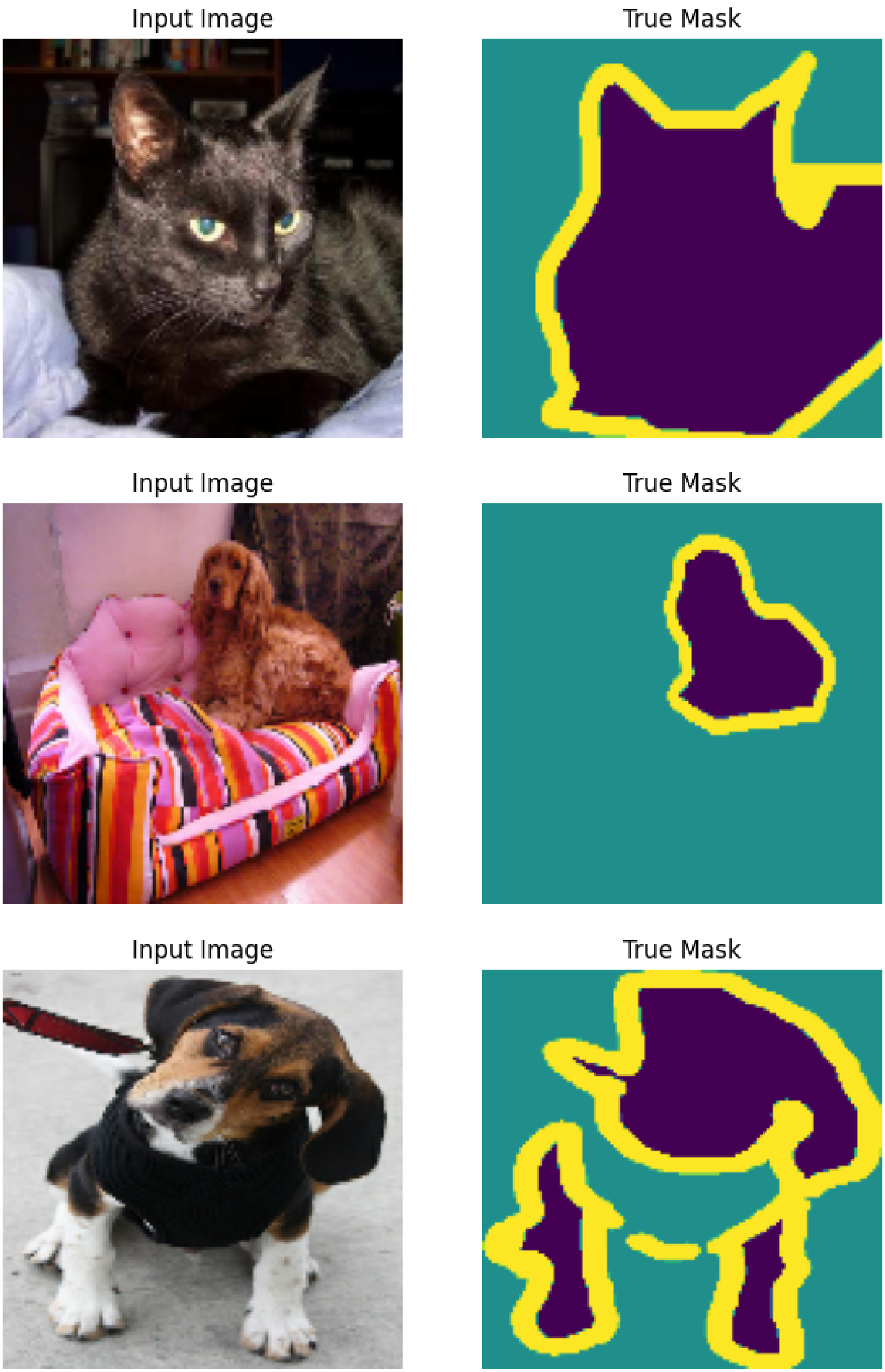
In [9]:
```
train_dataset = dataset['train'].map(load_train_images, num_parallel_calls=t
test_dataset = dataset['test'].map(load_test_images,num_parallel_calls=tf.da
```

In [10]:
```
BATCH_SIZE=64
BUFFER_SIZE=1000
train_dataset = train_dataset.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
train_dataset = train_dataset.prefetch(buffer_size=tf.data.experimental.AUTO
test_dataset = test_dataset.batch(BATCH_SIZE)
```

EDA

In [11]:
```
def display_sample(image_list):
  plt.figure(figsize=(8,8))
  title = ['Input Image', 'True Mask' , 'Predicted Mask']
  for i in range(len(image_list)):
    plt.subplot(1,len(image_list),i+1)
    plt.title(title[i])
    plt.imshow(tf.keras.preprocessing.image.array_to_img(image_list[i]))
    plt.axis('off')
  plt.show()
```

In [12]:
```
for images, masks in train_dataset.take(3):
  sample_image, sample_mask = images[0], masks[0]
  display_sample([sample_image, sample_mask])
```


Input Image
True Mask
Input Image
True Mask
Input Image
True Mask

In [12]:

```

```

Define U-Net Model

In [13]:

```
def double_conv_block(x, n_filters):
  x= Conv2D(n_filters, 3, padding='same', activation='relu', kernel_initiali
  x= Conv2D(n_filters, 3, padding='same', activation='relu', kernel_initiali
  return x

def downsample_block(x, n_filters):
  f = double_conv_block(x, n_filters)
  p = MaxPooling2D((2,2))(f)
  p = Dropout(0.3)(p)
  return f, p

def upsample_block(x,conv_features, n_filters):
  x= layers.Conv2DTranspose(n_filters,3,2, padding='same')(x)
  x= layers.Concatenate()([x, conv_features])
  x= Dropout(0.3)(x)
  x = double_conv_block(x, n_filters)
  return x
```

In [14]:

```
def build_unet_model(output_channels):
  inputs= layers.Input(shape=(128,128,3))
  f1, p1 = downsample_block(inputs, 64)
  f2, p2 = downsample_block(p1, 128)
  f3, p3 = downsample_block(p2, 256)
  f4, p4 = downsample_block(p3, 512)
  bottleneck = double_conv_block(p4, 1024)
  u6 = upsample_block(bottleneck, f4, 512)
  u7 = upsample_block(u6, f3, 256)
  u8 = upsample_block(u7, f2, 128)
  u9 = upsample_block(u8, f1, 64)
  outputs = layers.Conv2D(output_channels, 1, padding='same', activation='so
  unet_model = tf.keras.Model(inputs, outputs, name='U-Net')
  return unet_model
```

In [15]:

```
output_channels=3
model= build_unet_model(output_channels)
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metr
```

In [16]:

```
model.summary()
```

Model: "U-Net"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 128, 128, 3)	0	-
conv2d (Conv2D)	(None, 128, 128, 64)	1,792	input_layer[0
conv2d_1 (Conv2D)	(None, 128, 128, 64)	36,928	conv2d[0][0]
max_pooling2d (MaxPooling2D)	(None, 64, 64, 64)	0	conv2d_1[0][0
dropout (Dropout)	(None, 64, 64, 64)	0	max_pooling2d
conv2d_2 (Conv2D)	(None, 64, 64, 128)	73,856	dropout[0][0]
conv2d_3 (Conv2D)	(None, 64, 64, 128)	147,584	conv2d_2[0][0
max_pooling2d_1 (MaxPooling2D)	(None, 32, 32, 128)	0	conv2d_3[0][0
dropout_1 (Dropout)	(None, 32, 32, 128)	0	max_pooling2d
conv2d_4 (Conv2D)	(None, 32, 32, 256)	295,168	dropout_1[0][
conv2d_5 (Conv2D)	(None, 32, 32, 256)	590,080	conv2d_4[0][0
max_pooling2d_2 (MaxPooling2D)	(None, 16, 16, 256)	0	conv2d_5[0][0
dropout_2 (Dropout)	(None, 16, 16, 256)	0	max_pooling2d
conv2d_6 (Conv2D)	(None, 16, 16, 512)	1,180,160	dropout_2[0][
conv2d_7 (Conv2D)	(None, 16, 16, 512)	2,359,808	conv2d_6[0][0
max_pooling2d_3 (MaxPooling2D)	(None, 8, 8, 512)	0	conv2d_7[0][0
dropout_3 (Dropout)	(None, 8, 8, 512)	0	max_pooling2d
conv2d_8 (Conv2D)	(None, 8, 8, 1024)	4,719,616	dropout_3[0][
conv2d_9 (Conv2D)	(None, 8, 8, 1024)	9,438,208	conv2d_8[0][0
conv2d_transpose	(None, 16, 16,	4,719,104	conv2d_9[0][0

(Conv2DTranspose)	512)		
concatenate (Concatenate)	(None, 16, 16, 1024)	0	conv2d_transp conv2d_7[0][0
dropout_4 (Dropout)	(None, 16, 16, 1024)	0	concatenate[0
conv2d_10 (Conv2D)	(None, 16, 16, 512)	4,719,104	dropout_4[0][
conv2d_11 (Conv2D)	(None, 16, 16, 512)	2,359,808	conv2d_10[0][
conv2d_transpose_1 (Conv2DTranspose)	(None, 32, 32, 256)	1,179,904	conv2d_11[0][
concatenate_1 (Concatenate)	(None, 32, 32, 512)	0	conv2d_transp conv2d_5[0][0
dropout_5 (Dropout)	(None, 32, 32, 512)	0	concatenate_1
conv2d_12 (Conv2D)	(None, 32, 32, 256)	1,179,904	dropout_5[0][
conv2d_13 (Conv2D)	(None, 32, 32, 256)	590,080	conv2d_12[0][
conv2d_transpose_2 (Conv2DTranspose)	(None, 64, 64, 128)	295,040	conv2d_13[0][
concatenate_2 (Concatenate)	(None, 64, 64, 256)	0	conv2d_transp conv2d_3[0][0
dropout_6 (Dropout)	(None, 64, 64, 256)	0	concatenate_2
conv2d_14 (Conv2D)	(None, 64, 64, 128)	295,040	dropout_6[0][
conv2d_15 (Conv2D)	(None, 64, 64, 128)	147,584	conv2d_14[0][
conv2d_transpose_3 (Conv2DTranspose)	(None, 128, 128, 64)	73,792	conv2d_15[0][
concatenate_3 (Concatenate)	(None, 128, 128, 128)	0	conv2d_transp conv2d_1[0][0
dropout_7 (Dropout)	(None, 128, 128, 128)	0	concatenate_3
conv2d_16 (Conv2D)	(None, 128, 128, 64)	73,792	dropout_7[0][
conv2d_17 (Conv2D)	(None, 128, 128, 64)	36,928	conv2d_16[0][
conv2d_18 (Conv2D)	(None, 128, 128,	195	conv2d_17[0][

	3)		

Total params: 34,513,475 (131.66 MB)

Trainable params: 34,513,475 (131.66 MB)

Non-trainable params: 0 (0.00 B)

In [17]:

```
EPOCHS=20
steps_per_epoch = info.splits['train'].num_examples // BATCH_SIZE
validation_steps = info.splits['test'].num_examples // BATCH_SIZE

history = model.fit(train_dataset, epochs=EPOCHS, steps_per_epoch=steps_per_
```

```
Epoch 1/20
57/57 163s 1s/step - accuracy: 0.5696 - loss: 1.0117 -
val_accuracy: 0.5726 - val_loss: 0.8488
Epoch 2/20
57/57 123s 1s/step - accuracy: 0.5816 - loss: 0.8483 -
val_accuracy: 0.5971 - val_loss: 0.8044
Epoch 3/20
57/57 62s 1s/step - accuracy: 0.6104 - loss: 0.7942 - v
al_accuracy: 0.6256 - val_loss: 0.7931
Epoch 4/20
57/57 82s 1s/step - accuracy: 0.6329 - loss: 0.7790 - v
al_accuracy: 0.6856 - val_loss: 0.8682
Epoch 5/20
57/57 62s 1s/step - accuracy: 0.6423 - loss: 0.7791 - v
al_accuracy: 0.6271 - val_loss: 0.7735
Epoch 6/20
57/57 63s 1s/step - accuracy: 0.6336 - loss: 0.7783 - v
al_accuracy: 0.6809 - val_loss: 0.7202
Epoch 7/20
57/57 63s 1s/step - accuracy: 0.7128 - loss: 0.6705 - v
al_accuracy: 0.7364 - val_loss: 0.6243
Epoch 8/20
57/57 63s 1s/step - accuracy: 0.7458 - loss: 0.6089 - v
al_accuracy: 0.7725 - val_loss: 0.5510
Epoch 9/20
57/57 63s 1s/step - accuracy: 0.7725 - loss: 0.5518 - v
al_accuracy: 0.7851 - val_loss: 0.5161
Epoch 10/20
57/57 63s 1s/step - accuracy: 0.7878 - loss: 0.5163 - v
al_accuracy: 0.7965 - val_loss: 0.5034
Epoch 11/20
57/57 63s 1s/step - accuracy: 0.7987 - loss: 0.4934 - v
al_accuracy: 0.8134 - val_loss: 0.4621
Epoch 12/20
57/57 82s 1s/step - accuracy: 0.8150 - loss: 0.4521 - v
al_accuracy: 0.8228 - val_loss: 0.4412
Epoch 13/20
57/57 82s 1s/step - accuracy: 0.8246 - loss: 0.4323 - v
al_accuracy: 0.8193 - val_loss: 0.4484
Epoch 14/20
57/57 62s 1s/step - accuracy: 0.8278 - loss: 0.4237 - v
al_accuracy: 0.8407 - val_loss: 0.3947
Epoch 15/20
57/57 82s 1s/step - accuracy: 0.8404 - loss: 0.3906 - v
al_accuracy: 0.8447 - val_loss: 0.3830
Epoch 16/20
57/57 63s 1s/step - accuracy: 0.8474 - loss: 0.3716 - v
al_accuracy: 0.8459 - val_loss: 0.3779
Epoch 17/20
57/57 63s 1s/step - accuracy: 0.8486 - loss: 0.3705 - v
al_accuracy: 0.8508 - val_loss: 0.3768
Epoch 18/20
57/57 82s 1s/step - accuracy: 0.8521 - loss: 0.3595 - v
al_accuracy: 0.8482 - val_loss: 0.3790
Epoch 19/20
57/57 82s 1s/step - accuracy: 0.8613 - loss: 0.3372 - v
```

```
al_accuracy: 0.8537 - val_loss: 0.3676
Epoch 20/20
57/57 ━━━━━━━━━━━━━━━━━━━━ 62s 1s/step - accuracy: 0.8640 - loss: 0.3296 - v
al_accuracy: 0.8430 - val_loss: 0.4041
```

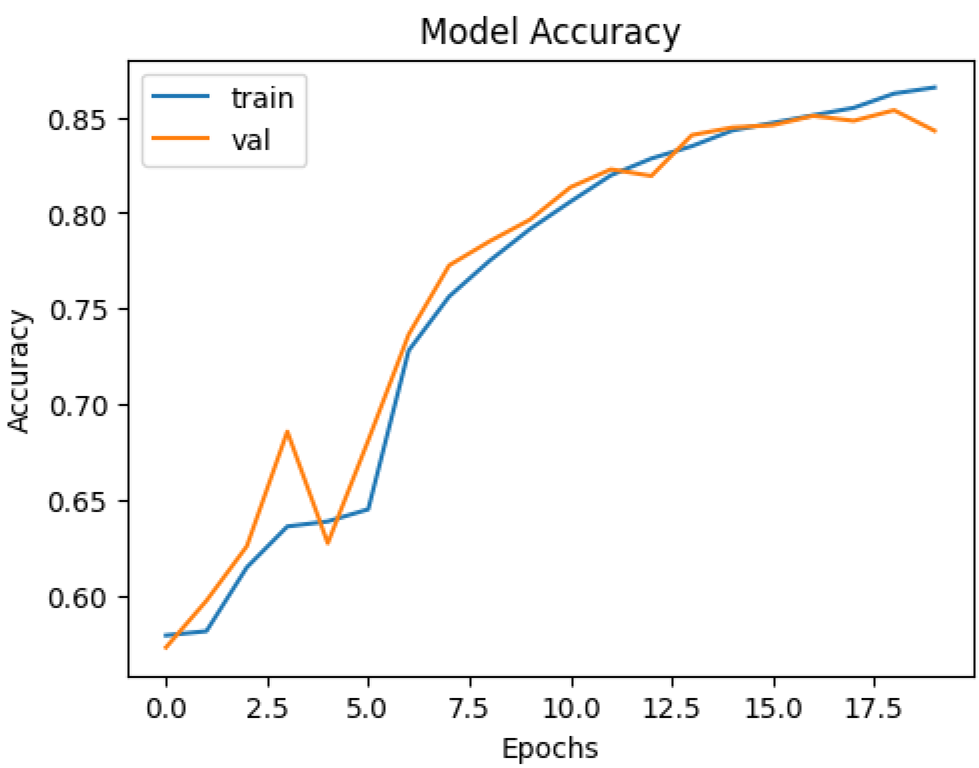
In [ ]:

Visualize the results

In [20]:

```
## plot train and val accuracy
plt.figure(figsize=(12,4))
plt.subplot(1,2,1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(['train', 'val'], loc='upper left')
```

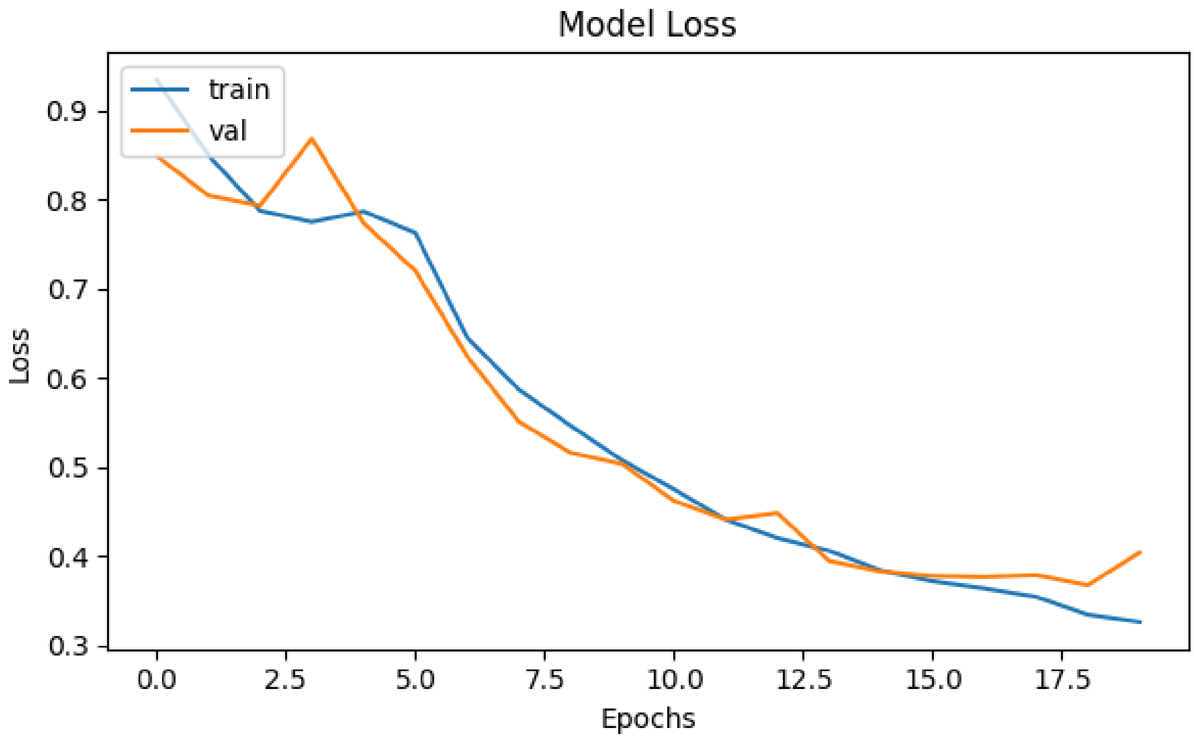
Out[20]: <matplotlib.legend.Legend at 0x7d9b37abacd0>



In [22]:

```
plt.figure(figsize=(12,4))
plt.subplot(1,2,1)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['train', 'val'], loc='upper left')
```

```
plt.tight_layout()
plt.show()
```



In [31]:
```
def create_mask(pred_msk):
  pred_mask = tf.argmax(pred_msk, axis=-1)
  pred_mask = pred_mask[..., tf.newaxis]
  return pred_mask[0]

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)]
      display_sample(display)
```

In [32]:
```
for image, mask in test_dataset.take(1):
  pred_mask = model.predict(image)

pred_mask = tf.argmax(pred_mask, axis=-1)
pred_mask = pred_mask[..., tf.newaxis]
print(pred_mask[0].shape)
```

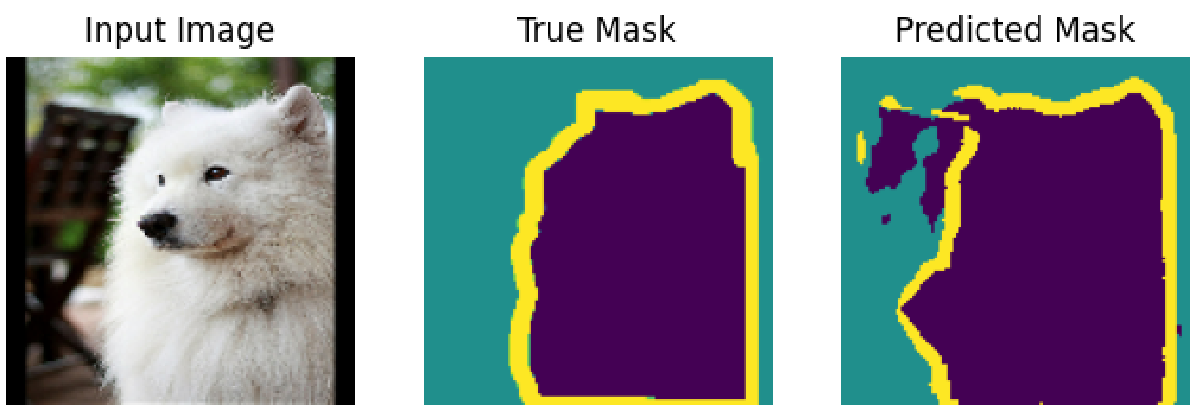
```
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 175ms/step
(128, 128, 1)
```

In [33]:
```
show_predictions(test_dataset,10)
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 182ms/step
```

2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 166ms/step

2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 167ms/step

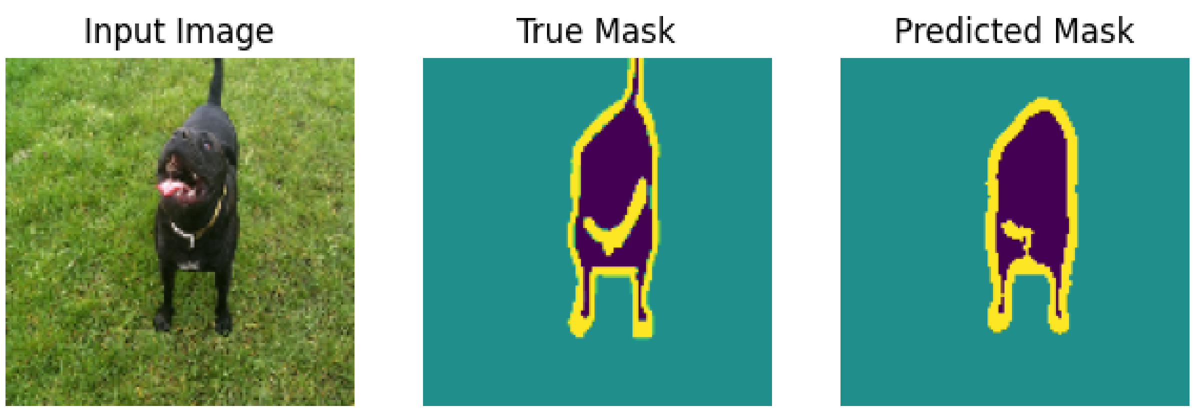


2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 165ms/step

2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 159ms/step

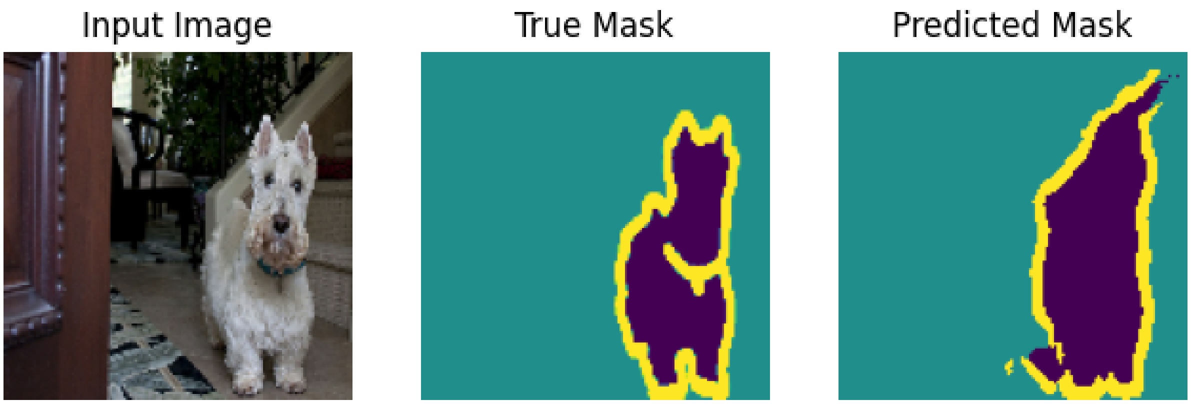
2/2 ━━━━━━━━━━━━━━━━━━━━ **0s** 158ms/step

2/2 ━━━━━━━━━━━━━━━━━━━━ **0s** 161ms/step



2/2 ━━━━━━━━━━━━━━━━━━━━ **0s** 161ms/step

2/2 ━━━━━━━━━━━━━━━━━━━━ **0s** 167ms/step




```
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 160ms/step
```

In [38]:
```
model.save('history.h5')
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
```

Model Evaluation

In [36]:
```
loss, accuracy = model.evaluate(test_dataset)
print(f"Test Loss: {loss}")
print(f"Test Accuracy: {accuracy}")
```

```
58/58 ━━━━━━━━━━━━━━━━━━━━ 26s 438ms/step - accuracy: 0.8434 - loss: 0.4016
Test Loss: 0.40391096472740173
Test Accuracy: 0.8430185914039612
```

In [37]:
```
from tensorflow.keras.metrics import MeanIoU

# Pick the number of classes (in Oxford-IIIT Pet, usually 3: pet, backgrounc
num_classes = 3
miou = MeanIoU(num_classes=num_classes)

for images, masks in test_dataset:
    preds = model.predict(images)
    preds = tf.argmax(preds, axis=-1)
    preds = preds[..., tf.newaxis]
    miou.update_state(masks, preds)
```

```
print("Mean IoU:", miou.result().numpy())
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 177ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 156ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 157ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 157ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 162ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 163ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 176ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 169ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 157ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 156ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 161ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 161ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 157ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 159ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 157ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 157ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 166ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 166ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 159ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 157ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 157ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 162ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 159ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 157ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 159ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 159ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 160ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 163ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 159ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 162ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 181ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 155ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 159ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 159ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 159ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 160ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 156ms/step
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 160ms/step
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 158ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 959ms/step
Mean IoU: 0.6517233
```

📊 Model Performance Summary

The U-Net-based image segmentation model trained on the Oxford-IIIT Pet Dataset demonstrates strong performance in pixel-wise classification:

Test Loss: 0.4039

Test Accuracy: 84.30%

Mean Intersection over Union (IoU): 0.65

These metrics indicate that the model has effectively learned to distinguish between the foreground (pet), background, and outlines, achieving a reliable segmentation quality across the validation samples.

✅ Conclusion

The U-Net architecture, known for its encoder-decoder structure and skip connections, proved to be highly effective for the task of semantic segmentation. With a test accuracy of over 84% and a mean IoU of 0.65, the model demonstrates a good balance between precision and recall in segmenting pet images. This validates the robustness of the chosen preprocessing pipeline and training strategy, making the model suitable for real-world image segmentation tasks where spatial accuracy is critical.

In [ ]: