

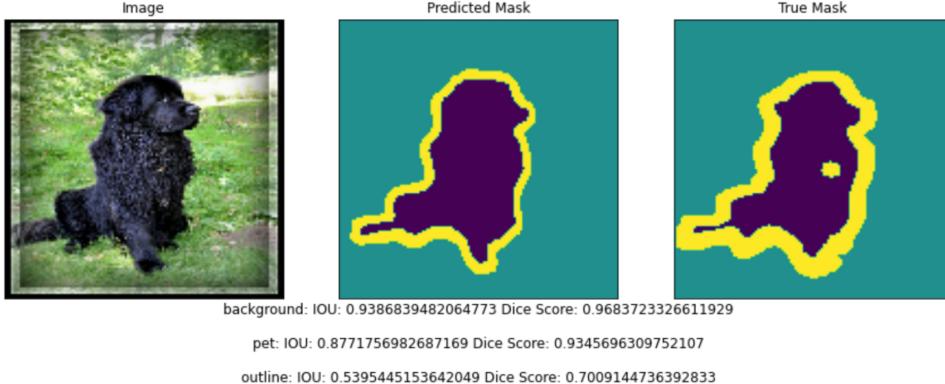


## Ungraded Lab: U-Net for Image Segmentation



This notebook illustrates how to build a [UNet](#) for semantic image segmentation. This architecture is also a fully convolutional network and is similar to the model you just built in the previous lesson. A key difference is the use of skip connections from the encoder to the decoder. You will see how this is implemented later as you build each part of the network.

At the end of this lab, you will be able to use the UNet to output segmentation masks that shows which pixels of an input image are part of the background, foreground, and outline.



### Imports

```
[1] try:  

    # %tensorflow_version only exists in Colab.  

    %tensorflow_version 2.x  

except Exception:  

    pass  

import tensorflow as tf  

import tensorflow_datasets as tfds  

import matplotlib.pyplot as plt  

import numpy as np
```

### Download the Oxford-IIIT Pets dataset

You will be training the model on the [Oxford Pets - IIIT dataset](#) dataset. This contains pet images, their classes, segmentation masks and head region-of-interest. You will only use the images and segmentation masks in this lab.

This dataset is already included in TensorFlow Datasets and you can simply download it. The segmentation masks are included in versions 3 and above. The cell below will download the dataset and place the results in a dictionary named `dataset`. It will also collect information about the dataset and we'll assign it to a variable named `info`.

```
[2] # If you hit a problem with checksums, you can execute the following line first  

!python -m tensorflow_datasets.scripts.download_and_prepare --register_checksums --datasets=oxford_iit_pet:3.1.0  

# download the dataset and get info  

dataset, info = tfds.load('oxford_iit_pet:3.*', with_info=True)  

2021-08-09 03:12:08.839910: I tensorflow/stream_executor/platform/default/dso_loader.cc:53] Successfully opened dynamic library libcuda.so.11.0  

I0809 03:12:11.170745 139844525442944 download_and_prepare.py:200] Running download_and_prepare for dataset(s):  

oxford_iit_pet:3.1.0  

Traceback (most recent call last):  

  File "/usr/lib/python3.7/runpy.py", line 193, in _run_module_as_main  

    "__main__", mod_spec)  

  File "/usr/lib/python3.7/runpy.py", line 85, in _run_code  

    exec(code, run_globals)  

  File "/usr/local/lib/python3.7/dist-packages/tensorflow_datasets/scripts/download_and_prepare.py", line 242, in <module>  

    app.run(main)  

  File "/usr/local/lib/python3.7/dist-packages/absl/app.py", line 303, in run  

    _run_main(main, args)  

  File "/usr/local/lib/python3.7/dist-packages/absl/app.py", line 251, in _run_main  

    sys.exit(main(argv))  

  File "/usr/local/lib/python3.7/dist-packages/tensorflow_datasets/scripts/download_and_prepare.py", line 203, in main  

    for name in datasets_to_build  

  File "/usr/local/lib/python3.7/dist-packages/tensorflow_datasets/scripts/download_and_prepare.py", line 203, in <dictcomp>  

    for name in datasets_to_build  

  File "/usr/local/lib/python3.7/dist-packages/tensorflow_datasets/core/load.py", line 207, in builder  

    return cls(data_dir=data_dir, **builder_kwargs, **builder_init_kwargs) # pytype: disable=not-instantiable  

  File "/usr/local/lib/python3.7/dist-packages/tensorflow_datasets/core/dataset_builder.py", line 183, in __init__  

    self._version = self._pick_version(version)  

  File "/usr/local/lib/python3.7/dist-packages/tensorflow_datasets/core/dataset_builder.py", line 241, in _pick_version  

    raise AssertionError(msg)  

AssertionError: Failed to construct dataset oxford_iit_pet:3.1.0: Dataset oxford_iit_pet cannot be loaded at version 3.1.0, only: 3.2.0.  

Downloading and preparing dataset oxford_iit_pet:3.2.0 (download: 773.52 MiB, generated: 774.69 MiB, total: 1.51 GiB) to /root/tensorflow_datasets/oxford_iit_pet/3.2.0...  

  Di Completed...: 100% [2/00:46<0:00, 23.32s/ url]  

  Di Size...: 100% [773/773 [00:46<0:00, 16.59 MiB/s]  

  Extraction completed...: 100% [2/00:46<0:00, 23.28s/ file]
```

```

Shuffling and writing examples to /root/tensorflow_datasets/oxford_iit_pet/3.2.0.incompleteZBF91R/oxford_iit_pet-train.tfrecord
93% ██████████ 3406/3680 [00:01<00:00, 2794.49 examples/s]
Shuffling and writing examples to /root/tensorflow_datasets/oxford_iit_pet/3.2.0.incompleteZBF91R/oxford_iit_pet-test.tfrecord
91% ██████████ 3341/3669 [00:09<00:00, 3659.28 examples/s]
Dataset oxford_iit_pet downloaded and prepared to /root/tensorflow_datasets/oxford_iit_pet/3.2.0. Subsequent calls will reuse this data.

```

Let's briefly examine the contents of the dataset you just downloaded.

```

[3] # see the possible keys we can access in the dataset dict.
# this contains the test and train splits.
print(dataset.keys())

dict_keys(['test', 'train'])

[4] # see information about the dataset
print(info)

tfds.core.DatasetInfo(
    name='oxford_iit_pet',
    version=3.2.0,
    description='The Oxford-IIIT pet dataset is a 37 category pet image dataset with roughly 200 images for each class. The images have large variations in scale, pose and lighting. All images have an associated ground truth annotation of breed.',
    homepage='http://www.robots.ox.ac.uk/~vgg/data/pets/',
    features=FeaturesDict({
        'file_name': Text(shape=(), dtype=tf.string),
        'image': Image(shape=(None, None, 3), dtype=tf.uint8),
        'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=37),
        'segmentation_mask': Image(shape=(None, None, 1), dtype=tf.uint8),
        'species': ClassLabel(shape=(), dtype=tf.int64, num_classes=2),
    }),
    total_num_examples=7349,
    splits={
        'test': 3669,
        'train': 3680,
    },
    supervised_keys=('image', 'label'),
    citation='''@InProceedings{parkhi12a,
        author = "Parkhi, O. M. and Vedaldi, A. and Zisserman, A. and Jawahar, C.~V.",
        title = "Cats and Dogs",
        booktitle = "IEEE Conference on Computer Vision and Pattern Recognition",
        year = "2012",
    }''',
    redistribution_info=
)

```

## ▼ Prepare the Dataset

You will now prepare the train and test sets. The following utility functions preprocess the data. These include:

- simple augmentation by flipping the image
- normalizing the pixel values
- resizing the images

Another preprocessing step is to adjust the segmentation mask's pixel values. The `README` in the [annotations](#) folder of the dataset mentions that the pixels in the segmentation mask are labeled as such:

Label	Class Name
1	foreground
2	background
3	Not Classified

For convenience, let's subtract 1 from these values and we will interpret these as `{'pet', 'background', 'outline'}`:

Label	Class Name
0	pet
1	background
2	outline

```

[5] # Preprocessing Utilities

def random_flip(input_image, input_mask):
    '''does a random flip of the image and mask'''
    if tf.random.uniform(() > 0.5:
        input_image = tf.image.flip_left_right(input_image)
        input_mask = tf.image.flip_left_right(input_mask)

    return input_image, input_mask

def normalize(input_image, input_mask):
    ...
    normalizes the input image pixel values to be from [0,1].
    subtracts 1 from the mask labels to have a range from [0,2]
    ...
    input_image = tf.cast(input_image, tf.float32) / 255.0
    input_mask -= 1
    return input_image, input_mask

@tf.function
def load_image_train(datapoint):
    '''resizes, normalizes, and flips the training data'''
    input_image = tf.image.resize(datapoint['image'], (128, 128), method='nearest')
    input_mask = tf.image.resize(datapoint['segmentation_mask'], (128, 128), method='nearest')
    input_image, input_mask = random_flip(input_image, input_mask)
    input_image, input_mask = normalize(input_image, input_mask)

```

```

    return input_image, input_mask

def load_image_test(datapoint):
    '''resizes and normalizes the test data'''
    input_image = tf.image.resize(datapoint['image'], (128, 128), method='nearest')
    input_mask = tf.image.resize(datapoint['segmentation_mask'], (128, 128), method='nearest')
    input_image, input_mask = normalize(input_image, input_mask)

    return input_image, input_mask

```

You can now call the utility functions above to prepare the train and test sets. The dataset you downloaded from TFDS already contains these splits and you will use those by simply accessing the `train` and `test` keys of the dataset dictionary.

Note: The `tf.data.experimental.AUTOTUNE` you see in this notebook is simply a constant equal to `-1`. This value is passed to allow certain methods to automatically set parameters based on available resources. For instance, `num_parallel_calls` parameter below will be set dynamically based on the available CPUs. The docstrings will show if a parameter can be autotuned. [Here](#) is the entry describing what it does to `num_parallel_calls`.

```

[6] # preprocess the train and test sets
train = dataset['train'].map(load_image_train, num_parallel_calls=tf.data.experimental.AUTOTUNE)
test = dataset['test'].map(load_image_test)

```

Now that the splits are loaded, you can then prepare batches for training and testing.

```

[7] BATCH_SIZE = 64
BUFFER_SIZE = 1000

# shuffle and group the train set into batches
train_dataset = train.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()

# do a prefetch to optimize processing
train_dataset = train_dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)

# group the test set into batches
test_dataset = test.batch(BATCH_SIZE)

```

Let's define a few more utilities to help us visualize our data and metrics.

```

[8] # class list of the mask pixels
class_names = ['pet', 'background', 'outline']

def display_with_metrics(display_list, iou_list, dice_score_list):
    '''displays a list of images/masks and overlays a list of IOU and Dice Scores'''

    metrics_by_id = [(idx, iou, dice_score) for idx, (iou, dice_score) in enumerate(zip(iou_list, dice_score_list)) if iou > 0.0]
    metrics_by_id.sort(key=lambda tup: tup[1], reverse=True) # sorts in place

    display_string_list = ["{}: IOU: {} Dice Score: {}".format(class_names[idx], iou, dice_score) for idx, iou, dice_score in metrics_by_id]
    display_string = "\n\n".join(display_string_list)

    display(display_list, ["Image", "Predicted Mask", "True Mask"], display_string=display_string)

def display(display_list,titles=[], display_string=None):
    '''displays a list of images/masks'''

    plt.figure(figsize=(15, 15))

    for i in range(len(display_list)):
        plt.subplot(1, len(display_list), i+1)
        plt.title(titles[i])
        plt.xticks([])
        plt.yticks([])
        if display_string and i == 1:
            plt.xlabel(display_string, fontsize=12)
        img_arr = tf.keras.preprocessing.image.array_to_img(display_list[i])
        plt.imshow(img_arr)

    plt.show()

def show_image_from_dataset(dataset):
    '''displays the first image and its mask from a dataset'''

    for image, mask in dataset.take(1):
        sample_image, sample_mask = image, mask
        display([sample_image, sample_mask], titles=["Image", "True Mask"])

def plot_metrics(metric_name, title, ylim=5):
    '''plots a given metric from the model history'''
    plt.title(title)
    plt.ylim(0,ylim)
    plt.plot(model_history.history[metric_name],color='blue',label=metric_name)
    plt.plot(model_history.history['val_' + metric_name],color='green',label='val_' + metric_name)

```

Finally, you can take a look at an image example and its corresponding mask from the dataset.

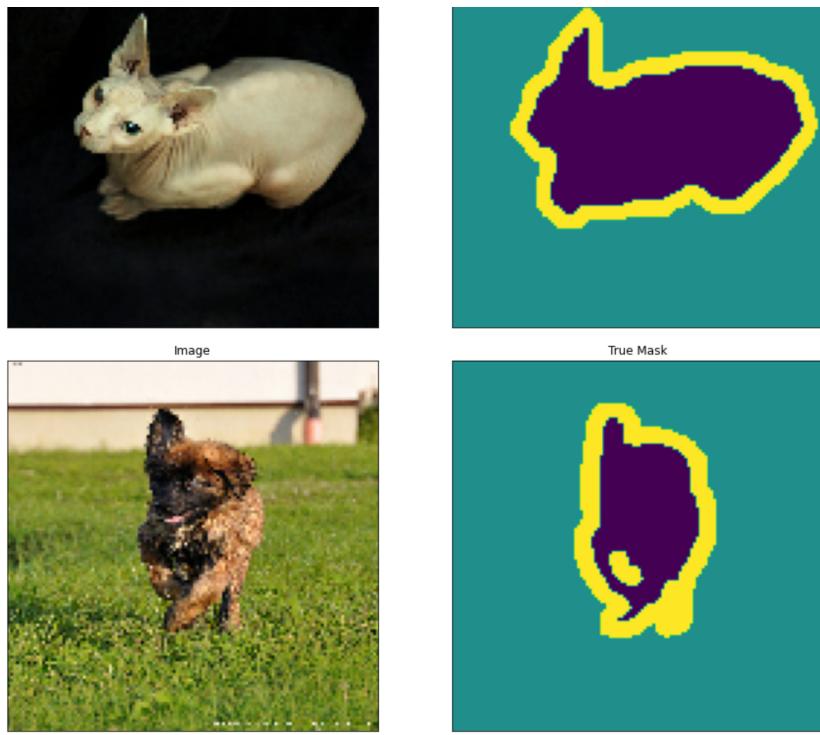
```

[9] # display an image from the train set
show_image_from_dataset(train)

# display an image from the test set
show_image_from_dataset(test)

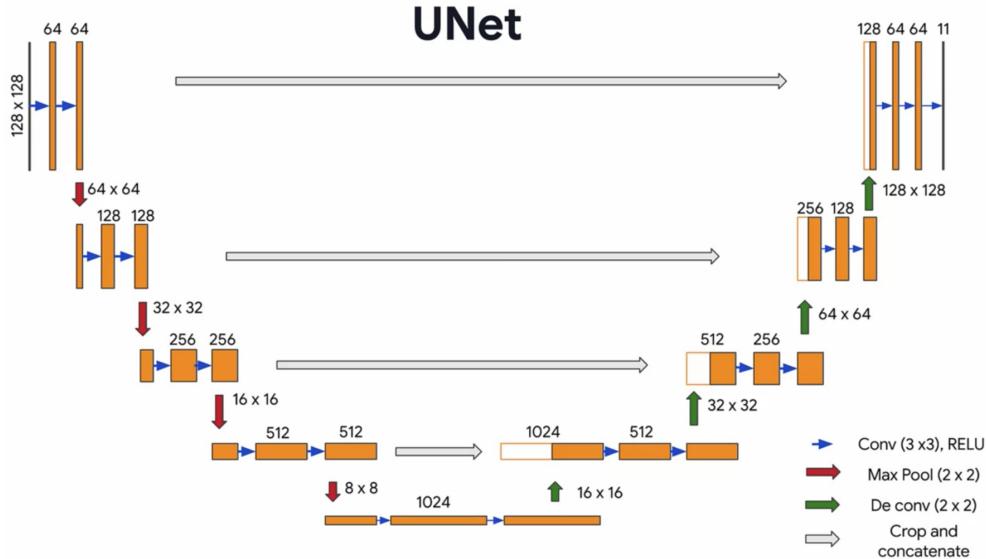
```





#### ▼ Define the model

With the dataset prepared, you can now build the UNet. Here is the overall architecture as shown in class:

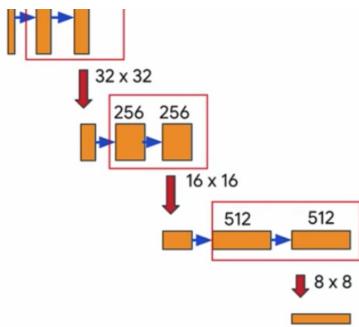


A UNet consists of an encoder (downsampler) and decoder (upsampler) with a bottleneck in between. The gray arrows correspond to the skip connections that concatenate encoder block outputs to each stage of the decoder. Let's see how to implement these starting with the encoder.

#### ▼ Encoder

Like the FCN model you built in the previous lesson, the encoder here will have repeating blocks (red boxes in the figure below) so it's best to create functions for it to make the code modular. These encoder blocks will contain two Conv2D layers activated by ReLU, followed by a MaxPooling and Dropout layer. As discussed in class, each stage will have increasing number of filters and the dimensionality of the features will reduce because of the pooling layer.





The encoder utilities will have three functions:

- `conv2d_block()` - to add two convolution layers and ReLU activations
- `encoder_block()` - to add pooling and dropout to the conv2d blocks. Recall that in UNet, you need to save the output of the convolution layers at each block so this function will return two values to take that into account (i.e. output of the conv block and the dropout)
- `encoder()` - to build the entire encoder. This will return the output of the last encoder block as well as the output of the previous conv blocks. These will be concatenated to the decoder blocks as you'll see later.

```
[10] # Encoder Utilities
0s
def conv2d_block(input_tensor, n_filters, kernel_size = 3):
    ...
    Adds 2 convolutional layers with the parameters passed to it

    Args:
        input_tensor (tensor) -- the input tensor
        n_filters (int) -- number of filters
        kernel_size (int) -- kernel size for the convolution

    Returns:
        tensor of output features
    ...

# first layer
x = input_tensor
for i in range(2):
    x = tf.keras.layers.Conv2D(filters = n_filters, kernel_size = (kernel_size, kernel_size), \
        kernel_initializer = 'he_normal', padding = 'same')(x)
    x = tf.keras.layers.Activation('relu')(x)

return x

def encoder_block(inputs, n_filters=64, pool_size=(2,2), dropout=0.3):
    ...
    Adds two convolutional blocks and then perform down sampling on output of convolutions.

    Args:
        input_tensor (tensor) -- the input tensor
        n_filters (int) -- number of filters
        kernel_size (int) -- kernel size for the convolution

    Returns:
        f - the output features of the convolution block
        p - the maxpooled features with dropout
    ...

    f = conv2d_block(inputs, n_filters=n_filters)
    p = tf.keras.layers.MaxPooling2D(pool_size=(2,2))(f)
    p = tf.keras.layers.Dropout(0.3)(p)

    return f, p

def encoder(inputs):
    ...
    This function defines the encoder or downsampling path.

    Args:
        inputs (tensor) -- batch of input images

    Returns:
        p4 - the output maxpooled features of the last encoder block
        (f1, f2, f3, f4) - the output features of all the encoder blocks
    ...
    f1, p1 = encoder_block(inputs, n_filters=64, pool_size=(2,2), dropout=0.3)
    f2, p2 = encoder_block(p1, n_filters=128, pool_size=(2,2), dropout=0.3)
    f3, p3 = encoder_block(p2, n_filters=256, pool_size=(2,2), dropout=0.3)
    f4, p4 = encoder_block(p3, n_filters=512, pool_size=(2,2), dropout=0.3)

    return p4, (f1, f2, f3, f4)
```

## Bottleneck

A bottleneck follows the encoder block and is used to extract more features. This does not have a pooling layer so the dimensionality remains the same. You can use the `conv2d_block()` function defined earlier to implement this.

```
[11] def bottleneck(inputs):
    ...
    This function defines the bottleneck convolutions to extract more features before the upsampling layers.
    ...

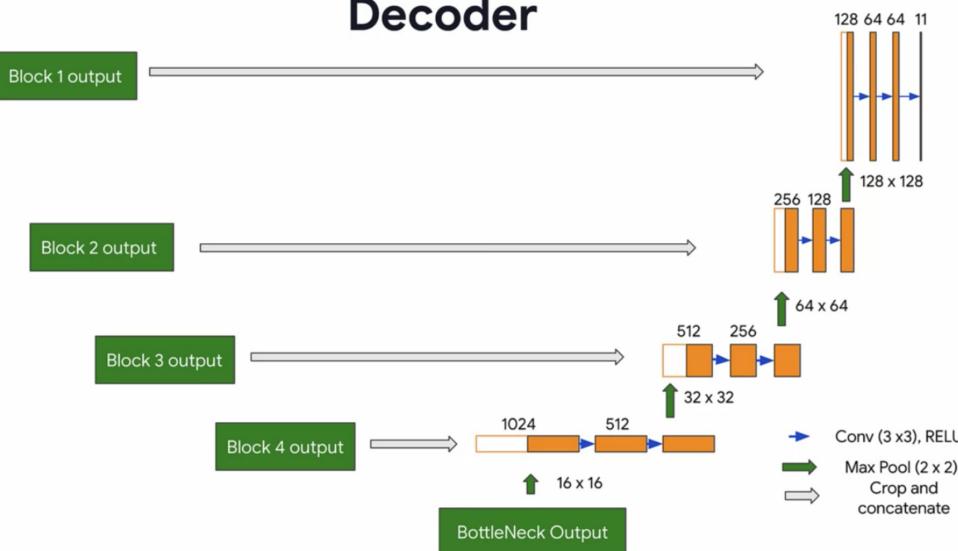
    bottle_neck = conv2d_block(inputs, n_filters=1024)
```

```
| return bottle_neck
```

## ▼ Decoder

Finally, we have the decoder which upsamples the features back to the original image size. At each upsampling level, you will take the output of the corresponding encoder block and concatenate it before feeding to the next decoder block. This is summarized in the figure below.

# Decoder



```
✓ [12] # Decoder Utilities
0s
def decoder_block(inputs, conv_output, n_filters=64, kernel_size=3, strides=3, dropout=0.3):
...
    defines the one decoder block of the UNet

    Args:
        inputs (tensor) -- batch of input features
        conv_output (tensor) -- features from an encoder block
        n_filters (int) -- number of filters
        kernel_size (int) -- kernel size
        strides (int) -- strides for the deconvolution/upsampling
        padding (string) -- "same" or "valid", tells if shape will be preserved by zero padding

    Returns:
        c (tensor) -- output features of the decoder block
...
    u = tf.keras.layers.Conv2DTranspose(n_filters, kernel_size, strides = strides, padding = 'same')(inputs)
    c = tf.keras.layers.concatenate([u, conv_output])
    c = tf.keras.layers.Dropout(dropout)(c)
    c = conv2d_block(c, n_filters, kernel_size=3)

    return c

def decoder(inputs, convs, output_channels):
...
    Defines the decoder of the UNet chaining together 4 decoder blocks.

    Args:
        inputs (tensor) -- batch of input features
        convs (tuple) -- features from the encoder blocks
        output_channels (int) -- number of classes in the label map

    Returns:
        outputs (tensor) -- the pixel wise label map of the image
...

    f1, f2, f3, f4 = convs

    c6 = decoder_block(inputs, f4, n_filters=512, kernel_size=(3,3), strides=(2,2), dropout=0.3)
    c7 = decoder_block(c6, f3, n_filters=256, kernel_size=(3,3), strides=(2,2), dropout=0.3)
    c8 = decoder_block(c7, f2, n_filters=128, kernel_size=(3,3), strides=(2,2), dropout=0.3)
    c9 = decoder_block(c8, f1, n_filters=64, kernel_size=(3,3), strides=(2,2), dropout=0.3)

    outputs = tf.keras.layers.Conv2D(output_channels, (1, 1), activation='softmax')(c9)

    return outputs
```

## ▼ Putting it all together

You can finally build the UNet by chaining the encoder, bottleneck, and decoder. You will specify the number of output channels and in this particular set, that would be 3. That is because there are three possible labels for each pixel: 'pet', 'background', and 'outline'.

```
✓ [13] OUTPUT_CHANNELS = 3
0s
def unet():
...
    Defines the UNet by connecting the encoder, bottleneck and decoder.
...

    # specify the input shape
    inputs = tf.keras.layers.Input(shape=(128, 128,3,))
```

```

# feed the inputs to the encoder
encoder_output, convs = encoder(inputs)

# feed the encoder output to the bottleneck
bottle_neck = bottleneck(encoder_output)

# feed the bottleneck and encoder block outputs to the decoder
# specify the number of classes via the `output_channels` argument
outputs = decoder(bottle_neck, convs, output_channels=OUTPUT_CHANNELS)

# create the model
model = tf.keras.Model(inputs=inputs, outputs=outputs)

return model

# instantiate the model
model = unet()

# see the resulting model architecture
model.summary()

```

conv2d_10 (Conv2D)	(None, 16, 16, 512)	4719104	dropout_4[0][0]
activation_10 (Activation)	(None, 16, 16, 512)	0	conv2d_10[0][0]
conv2d_11 (Conv2D)	(None, 16, 16, 512)	2359808	activation_10[0][0]
activation_11 (Activation)	(None, 16, 16, 512)	0	conv2d_11[0][0]
conv2d_transpose_1 (Conv2DTrans)	(None, 32, 32, 256)	1179904	activation_11[0][0]
concatenate_1 (Concatenate)	(None, 32, 32, 512)	0	conv2d_transpose_1[0][0] activation_5[0][0]
dropout_5 (Dropout)	(None, 32, 32, 512)	0	concatenate_1[0][0]
conv2d_12 (Conv2D)	(None, 32, 32, 256)	1179904	dropout_5[0][0]
activation_12 (Activation)	(None, 32, 32, 256)	0	conv2d_12[0][0]
conv2d_13 (Conv2D)	(None, 32, 32, 256)	590080	activation_12[0][0]
activation_13 (Activation)	(None, 32, 32, 256)	0	conv2d_13[0][0]
conv2d_transpose_2 (Conv2DTrans)	(None, 64, 64, 128)	295040	activation_13[0][0]
concatenate_2 (Concatenate)	(None, 64, 64, 256)	0	conv2d_transpose_2[0][0] activation_3[0][0]
dropout_6 (Dropout)	(None, 64, 64, 256)	0	concatenate_2[0][0]
conv2d_14 (Conv2D)	(None, 64, 64, 128)	295040	dropout_6[0][0]
activation_14 (Activation)	(None, 64, 64, 128)	0	conv2d_14[0][0]
conv2d_15 (Conv2D)	(None, 64, 64, 128)	147584	activation_14[0][0]
activation_15 (Activation)	(None, 64, 64, 128)	0	conv2d_15[0][0]
conv2d_transpose_3 (Conv2DTrans)	(None, 128, 128, 64)	73792	activation_15[0][0]
concatenate_3 (Concatenate)	(None, 128, 128, 128)	0	conv2d_transpose_3[0][0] activation_1[0][0]
dropout_7 (Dropout)	(None, 128, 128, 128)	0	concatenate_3[0][0]
conv2d_16 (Conv2D)	(None, 128, 128, 64)	73792	dropout_7[0][0]
activation_16 (Activation)	(None, 128, 128, 64)	0	conv2d_16[0][0]
conv2d_17 (Conv2D)	(None, 128, 128, 64)	36928	activation_16[0][0]
activation_17 (Activation)	(None, 128, 128, 64)	0	conv2d_17[0][0]
conv2d_18 (Conv2D)	(None, 128, 128, 3)	195	activation_17[0][0]
=====			
Total params: 34,513,475			
Trainable params: 34,513,475			
Non-trainable params: 0			

## Compile and Train the model

Now, all that is left to do is to compile and train the model. The loss you will use is `sparse_categorical_crossentropy`. The reason is because the network is trying to assign each pixel a label, just like multi-class prediction. In the true segmentation mask, each pixel has either a {0,1,2}. The network here is outputting three channels. Essentially, each channel is trying to learn to predict a class and `sparse_categorical_crossentropy` is the recommended loss for such a scenario.

```
[14] # configure the optimizer, loss and metrics for training
model.compile(optimizer=tf.keras.optimizers.Adam(),
               loss='sparse_categorical_crossentropy',
               metrics=['accuracy'])
```

```
[15] # configure the training parameters and train the model

TRAIN_LENGTH = info.splits['train'].num_examples
EPOCHS = 10
VAL_SUBSPLITS = 5
STEPS_PER_EPOCH = TRAIN_LENGTH // BATCH_SIZE
VALIDATION_STEPS = info.splits['test'].num_examples//BATCH_SIZE//VAL_SUBSPLITS

# this will take around 20 minutes to run
model_history = model.fit(train_dataset, epochs=EPOCHS,
                           steps_per_epoch=STEPS_PER_EPOCH,
                           validation_steps=VALIDATION_STEPS,
                           validation_data=test_dataset)
```

```
Epoch 1/10
57/57 [=====] - 100s 845ms/step - loss: 0.9081 - accuracy: 0.5793 - val_loss: 0.8351 - val_accuracy: 0.5791
Epoch 2/10
57/57 [=====] - 58s 860ms/step - loss: 0.7692 - accuracy: 0.6484 - val_loss: 0.8529 - val_accuracy: 0.6524
```

```

Epoch 3/10
57/57 [=====] - 49s 869ms/step - loss: 0.6620 - accuracy: 0.7283 - val_loss: 0.5939 - val_accuracy: 0.7634
Epoch 4/10
57/57 [=====] - 50s 880ms/step - loss: 0.5877 - accuracy: 0.7684 - val_loss: 0.5436 - val_accuracy: 0.7843
Epoch 5/10
57/57 [=====] - 50s 886ms/step - loss: 0.5240 - accuracy: 0.7950 - val_loss: 0.5646 - val_accuracy: 0.7700
Epoch 6/10
57/57 [=====] - 51s 887ms/step - loss: 0.4941 - accuracy: 0.8086 - val_loss: 0.4530 - val_accuracy: 0.8268
Epoch 7/10
57/57 [=====] - 51s 892ms/step - loss: 0.4509 - accuracy: 0.8260 - val_loss: 0.4578 - val_accuracy: 0.8278
Epoch 8/10
57/57 [=====] - 51s 895ms/step - loss: 0.4296 - accuracy: 0.8342 - val_loss: 0.4032 - val_accuracy: 0.8453
Epoch 9/10
57/57 [=====] - 51s 896ms/step - loss: 0.4130 - accuracy: 0.8415 - val_loss: 0.3883 - val_accuracy: 0.8507
Epoch 10/10
57/57 [=====] - 51s 900ms/step - loss: 0.3889 - accuracy: 0.8511 - val_loss: 0.3760 - val_accuracy: 0.8568

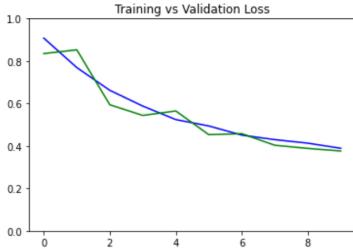
```

You can plot the train and validation loss to see how the training went. This should show generally decreasing values per epoch.

```

[16] # Plot the training and validation loss
plot_metrics("loss", title="Training vs Validation Loss", ylim=1)

```



## ▼ Make predictions

The model is now ready to make some predictions. You will use the test dataset you prepared earlier to feed input images that the model has not seen before. The utilities below will help in processing the test dataset and model predictions.

```

[17] # Prediction Utilities
def get_test_image_and_annotation_arrays():
    """
    Unpacks the test dataset and returns the input images and segmentation masks
    """

    ds = test_dataset.unbatch()
    ds = ds.batch(info.splits['test'].num_examples)

    images = []
    y_true_segments = []

    for image, annotation in ds.take(1):
        y_true_segments = annotation.numpy()
        images = image.numpy()

    y_true_segments = y_true_segments[: (info.splits['test'].num_examples - (info.splits['test'].num_examples % BATCH_SIZE))]

    return images[: (info.splits['test'].num_examples - (info.splits['test'].num_examples % BATCH_SIZE))], y_true_segments

def create_mask(pred_mask):
    """
    Creates the segmentation mask by getting the channel with the highest probability. Remember that we
    have 3 channels in the output of the UNet. For each pixel, the prediction will be the channel with the
    highest probability.
    """
    pred_mask = tf.argmax(pred_mask, axis=-1)
    pred_mask = pred_mask[..., tf.newaxis]
    return pred_mask[0].numpy()

def make_predictions(image, mask, num=1):
    """
    Feeds an image to a model and returns the predicted mask.
    """

    image = np.reshape(image,(1, image.shape[0], image.shape[1], image.shape[2]))
    pred_mask = model.predict(image)
    pred_mask = create_mask(pred_mask)

    return pred_mask

```

## ▼ Compute class wise metrics

Like the previous lab, you will also want to compute the IOU and Dice Score. This is the same function you used previously.

```

[18] def class_wise_metrics(y_true, y_pred):
    class_wise_iou = []
    class_wise_dice_score = []

    smoothening_factor = 0.00001
    for i in range(3):

        intersection = np.sum((y_pred == i) * (y_true == i))
        y_true_area = np.sum((y_true == i))
        y_pred_area = np.sum((y_pred == i))
        combined_area = y_true_area + y_pred_area

```

```

iou = (intersection + smoothening_factor) / (combined_area - intersection + smoothening_factor)
class_wise_iou.append(iou)

dice_score = 2 * ((intersection + smoothening_factor) / (combined_area + smoothening_factor))
class_wise_dice_score.append(dice_score)

return class_wise_iou, class_wise_dice_score

```

With all the utilities defined, you can now proceed to showing the metrics and feeding test images.

```

✓ [19] # Setup the ground truth and predictions.
27s
# get the ground truth from the test set
y_true_images, y_true_segments = get_test_image_and_annotation_arrays()

# feed the test set to the model to get the predicted masks
results = model.predict(test_dataset, steps=info.splits['test'].num_examples//BATCH_SIZE)
results = np.argmax(results, axis=3)
results = results[..., tf.newaxis]

✓ [20] # compute the class wise metrics
cls_wise_iou, cls_wise_dice_score = class_wise_metrics(y_true_segments, results)

✓ [21] # show the IOU for each class
0s
for idx, iou in enumerate(cls_wise_iou):
    spaces = ' ' * (10-len(class_names[idx]) + 2)
    print("{}{}{} ".format(class_names[idx], spaces, iou))

pet      0.7308897280239206
background 0.8442492083560594
outline   0.3750439488343814

✓ [22] # show the Dice Score for each class
0s
for idx, dice_score in enumerate(cls_wise_dice_score):
    spaces = ' ' * (10-len(class_names[idx]) + 2)
    print("{}{}{} ".format(class_names[idx], spaces, dice_score))

pet      0.844524889356807
background 0.915547860376996
outline   0.5455010352979837

```

#### ▼ Show Predictions

```

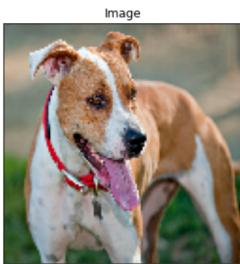
✓ 1s
# Please input a number between 0 to 3647 to pick an image from the dataset
integer_slider = 3646

# Get the prediction mask
y_pred_mask = make_predictions(y_true_images[integer_slider], y_true_segments[integer_slider])

# Compute the class wise metrics
iou, dice_score = class_wise_metrics(y_true_segments[integer_slider], y_pred_mask)

# Overlay the metrics with the images
display_with_metrics([y_true_images[integer_slider], y_pred_mask, y_true_segments[integer_slider]], iou, dice_score)

```



pet: IOU: 0.7880090500135645 Dice Score: 0.8814374295321793

background: IOU: 0.7487179490759004 Dice Score: 0.8563049862688946

outline: IOU: 0.418710265156796 Dice Score: 0.590268889051923

That's all for this lab! In the next section, you will learn about another type of image segmentation model: Mask R-CNN for instance segmentation!