Ungraded Lab: Fully Convolutional Neural Networks for Image Segmentation

This notebook illustrates how to build a Fully Convolutional Neural Network for semantic image segmentation.

You will train the model on a [custom dataset]() prepared by [divamgupta](). This contains video frames from a moving vehicle and is a subsample of the [CamVid]() dataset.

You will be using a pretrained VGG-16 network for the feature extraction path, then followed by an FCN-8 network for upsampling and generating the predictions. The output will be a label map (i.e. segmentation mask) with predictions for 12 classes. Let's begin!

Imports

```
import os
import zipfile
import PIL.Image, PIL.ImageFont, PIL.ImageDraw
import numpy as np

try:
  # %tensorflow_version only exists in Colab.
  %tensorflow_version 2.x
except Exception:
  pass

import tensorflow as tf
from matplotlib import pyplot as plt
import tensorflow_datasets as tfds
import seaborn as sns

print("Tensorflow version " + tf.__version__)
```

```
     Tensorflow version 2.4.1
```

Download the Dataset

We hosted the dataset in a Google bucket so you will need to download it first and unzip to a local directory.

```
# download the dataset (zipped file)
!wget --no-check-certificate \
    https://storage.googleapis.com/laurencemoroney-blog.appspot.com/fcnn-dataset.zip \
    -O /tmp/fcnn-dataset.zip

# extract the downloaded dataset to a local directory: /tmp/fcnn
local_zip = '/tmp/fcnn-dataset.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
```

```
zip_ref.extractall('/tmp/fcnn')
zip_ref.close()
```

```
--2021-05-22 20:18:33--  https://storage.googleapis.com/laurencemoroney-blog.appspot.com/fcnn-
Resolving storage.googleapis.com (storage.googleapis.com)... 74.125.142.128, 74.125.195.128, 1
Connecting to storage.googleapis.com (storage.googleapis.com)|74.125.142.128|:443... connected
HTTP request sent, awaiting response... 200 OK
Length: 125577577 (120M) [application/zip]
Saving to: '/tmp/fcnn-dataset.zip'

/tmp/fcnn-dataset.z 100%[===================>] 119.76M   105MB/s    in 1.1s

2021-05-22 20:18:34 (105 MB/s) - '/tmp/fcnn-dataset.zip' saved [125577577/125577577]
```

The dataset you just downloaded contains folders for images and annotations. The *images* contain the video frames while the *annotations* contain the pixel-wise label maps. Each label map has the shape `(height, width , 1)` with each point in this space denoting the corresponding pixel's class. Classes are in the range `[0, 11]` (i.e. 12 classes) and the pixel labels correspond to these classes:

Value	Class Name
0	sky
1	building
2	column/pole
3	road
4	side walk
5	vegetation
6	traffic light
7	fence
8	vehicle
9	pedestrian
10	byciclist
11	void

For example, if a pixel is part of a road, then that point will be labeled `3` in the label map. Run the cell below to create a list containing the class names:

- Note: bicyclist is mispelled as 'byciclist' in the dataset. We won't handle data cleaning in this example, but you can inspect and clean the data if you want to use this as a starting point for a personal project.

```
# pixel labels in the video frames
class_names = ['sky', 'building','column/pole', 'road', 'side walk', 'vegetation', 'traffic light',
```

Load and Prepare the Dataset

Next, you will load and prepare the train and validation sets for training. There are some preprocessing steps needed before the data is fed to the model. These include:

- resizing the height and width of the input images and label maps (224 x 224px by default)

- normalizing the input images' pixel values to fall in the range `[-1, 1]`
- reshaping the label maps from `(height, width, 1)` to `(height, width, 12)` with each slice along the third axis having `1` if it belongs to the class corresponding to that slice's index else `0`. For example, if a pixel is part of a road, then using the table above, that point at slice #3 will be labeled `1` and it will be `0` in all other slices. To illustrate using simple arrays:

```
# if we have a label map with 3 classes...
n_classes = 3
# and this is the original annotation...
orig_anno = [0 1 2]
# then the reshaped annotation will have 3 slices and its contents will look like this:
reshaped_anno = [1 0 0][0 1 0][0 0 1]
```

The following function will do the preprocessing steps mentioned above.

```
def map_filename_to_image_and_mask(t_filename, a_filename, height=224, width=224):
  '''
  Preprocesses the dataset by:
    * resizing the input image and label maps
    * normalizing the input image pixels
    * reshaping the label maps from (height, width, 1) to (height, width, 12)

  Args:
    t_filename (string) -- path to the raw input image
    a_filename (string) -- path to the raw annotation (label map) file
    height (int) -- height in pixels to resize to
    width (int) -- width in pixels to resize to

  Returns:
    image (tensor) -- preprocessed image
    annotation (tensor) -- preprocessed annotation
  '''

  # Convert image and mask files to tensors
  img_raw = tf.io.read_file(t_filename)
  anno_raw = tf.io.read_file(a_filename)
  image = tf.image.decode_jpeg(img_raw)
  annotation = tf.image.decode_jpeg(anno_raw)

  # Resize image and segmentation mask
  image = tf.image.resize(image, (height, width,))
  annotation = tf.image.resize(annotation, (height, width,))
  image = tf.reshape(image, (height, width, 3,))
  annotation = tf.cast(annotation, dtype=tf.int32)
  annotation = tf.reshape(annotation, (height, width, 1,))
  stack_list = []

  # Reshape segmentation masks
  for c in range(len(class_names)):
    mask = tf.equal(annotation[:,:,0], tf.constant(c))
    stack_list.append(tf.cast(mask, dtype=tf.int32))

  annotation = tf.stack(stack_list, axis=2)
```

```
  # Normalize pixels in the input image
  image = image/127.5
  image -= 1

  return image, annotation
```

The dataset also already has separate folders for train and test sets. As described earlier, these sets will have two folders: one corresponding to the images, and the other containing the annotations.

```
# show folders inside the dataset you downloaded
!ls /tmp/fcnn/dataset1
```

```
     annotations_prepped_test   images_prepped_test
     annotations_prepped_train  images_prepped_train
```

You will use the following functions to create the tensorflow datasets from the images in these folders. Notice that before creating the batches in the `get_training_dataset()` and `get_validation_set()`, the images are first preprocessed using the `map_filename_to_image_and_mask()` function you defined earlier.

```
# Utilities for preparing the datasets

BATCH_SIZE = 64

def get_dataset_slice_paths(image_dir, label_map_dir):
  '''
  generates the lists of image and label map paths

  Args:
    image_dir (string) -- path to the input images directory
    label_map_dir (string) -- path to the label map directory

  Returns:
    image_paths (list of strings) -- paths to each image file
    label_map_paths (list of strings) -- paths to each label map
  '''
  image_file_list = os.listdir(image_dir)
  label_map_file_list = os.listdir(label_map_dir)
  image_paths = [os.path.join(image_dir, fname) for fname in image_file_list]
  label_map_paths = [os.path.join(label_map_dir, fname) for fname in label_map_file_list]

  return image_paths, label_map_paths


def get_training_dataset(image_paths, label_map_paths):
  '''
  Prepares shuffled batches of the training set.

  Args:
    image_paths (list of strings) -- paths to each image file in the train set
    label_map_paths (list of strings) -- paths to each label map in the train set

  Returns:
    tf Dataset containing the preprocessed train set
  '''
  training_dataset = tf.data.Dataset.from_tensor_slices((image_paths, label_map_paths))
```

```
  training_dataset = training_dataset.map(map_filename_to_image_and_mask)
  training_dataset = training_dataset.shuffle(100, reshuffle_each_iteration=True)
  training_dataset = training_dataset.batch(BATCH_SIZE)
  training_dataset = training_dataset.repeat()
  training_dataset = training_dataset.prefetch(-1)

  return training_dataset


def get_validation_dataset(image_paths, label_map_paths):
  '''
  Prepares batches of the validation set.

  Args:
    image_paths (list of strings) -- paths to each image file in the val set
    label_map_paths (list of strings) -- paths to each label map in the val set

  Returns:
    tf Dataset containing the preprocessed validation set
  '''
  validation_dataset = tf.data.Dataset.from_tensor_slices((image_paths, label_map_paths))
  validation_dataset = validation_dataset.map(map_filename_to_image_and_mask)
  validation_dataset = validation_dataset.batch(BATCH_SIZE)
  validation_dataset = validation_dataset.repeat()

  return validation_dataset
```

You can now generate the training and validation sets by running the cell below.

```
# get the paths to the images
training_image_paths, training_label_map_paths = get_dataset_slice_paths('/tmp/fcnn/dataset1/images_
validation_image_paths, validation_label_map_paths = get_dataset_slice_paths('/tmp/fcnn/dataset1/ima

# generate the train and val sets
training_dataset = get_training_dataset(training_image_paths, training_label_map_paths)
validation_dataset = get_validation_dataset(validation_image_paths, validation_label_map_paths)
```

Let's Take a Look at the Dataset

You will also need utilities to help visualize the dataset and the model predictions later. First, you need to assign a color mapping to the classes in the label maps. Since our dataset has 12 classes, you need to have a list of 12 colors. We can use the color_palette() from Seaborn to generate this.

```
# generate a list that contains one color for each class
colors = sns.color_palette(None, len(class_names))

# print class name - normalized RGB tuple pairs
# the tuple values will be multiplied by 255 in the helper functions later
# to convert to the (0,0,0) to (255,255,255) RGB values you might be familiar with
for class_name, color in zip(class_names, colors):
  print(f'{class_name} -- {color}')
```

```
    sky -- (0.12156862745098039, 0.4666666666666667, 0.7058823529411765)
    building -- (1.0, 0.4980392156862745, 0.054901960784313725)
    column/pole -- (0.17254901960784313, 0.6274509803921569, 0.17254901960784313)
    road -- (0.8392156862745098, 0.15294117647058825, 0.1568627450980392)
    side walk -- (0.5803921568627451, 0.403921568627451, 0.7411764705882353)
    vegetation -- (0.5490196078431373, 0.33725490196078434, 0.29411764705882354)
    traffic light -- (0.8901960784313725, 0.4666666666666667, 0.7607843137254902)
    fence -- (0.4980392156862745, 0.4980392156862745, 0.4980392156862745)
    vehicle -- (0.7372549019607844, 0.7411764705882353, 0.13333333333333333)
    pedestrian -- (0.09019607843137255, 0.7450980392156863, 0.8117647058823529)
    byciclist -- (0.12156862745098039, 0.4666666666666667, 0.7058823529411765)
    void -- (1.0, 0.4980392156862745, 0.054901960784313725)


# Visualization Utilities

def fuse_with_pil(images):
  '''
  Creates a blank image and pastes input images

  Args:
    images (list of numpy arrays) - numpy array representations of the images to paste

  Returns:
    PIL Image object containing the images
  '''

  widths = (image.shape[1] for image in images)
  heights = (image.shape[0] for image in images)
  total_width = sum(widths)
  max_height = max(heights)

  new_im = PIL.Image.new('RGB', (total_width, max_height))

  x_offset = 0
  for im in images:
    pil_image = PIL.Image.fromarray(np.uint8(im))
    new_im.paste(pil_image, (x_offset,0))
    x_offset += im.shape[1]

  return new_im


def give_color_to_annotation(annotation):
  '''
  Converts a 2-D annotation to a numpy array with shape (height, width, 3) where
  the third axis represents the color channel. The label values are multiplied by
  255 and placed in this axis to give color to the annotation

  Args:
    annotation (numpy array) - label map array

  Returns:
    the annotation array with an additional color channel/axis
  '''
  seg_img = np.zeros( (annotation.shape[0],annotation.shape[1], 3) ).astype('float')

  for c in range(12):
    segc = (annotation == c)
    seg_img[:,:,0] += segc*( colors[c][0] * 255.0)
    seg_img[:,:,1] += segc*( colors[c][1] * 255.0)
```

```
    seg_img[:,:,1] += segc*( colors[c][1] * 255.0)
    seg_img[:,:,2] += segc*( colors[c][2] * 255.0)

  return seg_img


def show_predictions(image, labelmaps, titles, iou_list, dice_score_list):
  '''
  Displays the images with the ground truth and predicted label maps

  Args:
    image (numpy array) -- the input image
    labelmaps (list of arrays) -- contains the predicted and ground truth label maps
    titles (list of strings) -- display headings for the images to be displayed
    iou_list (list of floats) -- the IOU values for each class
    dice_score_list (list of floats) -- the Dice Score for each vlass
  '''

  true_img = give_color_to_annotation(labelmaps[1])
  pred_img = give_color_to_annotation(labelmaps[0])

  image = image + 1
  image = image * 127.5
  images = np.uint8([image, pred_img, true_img])

  metrics_by_id = [(idx, iou, dice_score) for idx, (iou, dice_score) in enumerate(zip(iou_list, dice
  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
  display_string = "\n\n".join(display_string_list)

  plt.figure(figsize=(15, 4))

  for idx, im in enumerate(images):
    plt.subplot(1, 3, idx+1)
    if idx == 1:
      plt.xlabel(display_string)
    plt.xticks([])
    plt.yticks([])
    plt.title(titles[idx], fontsize=12)
    plt.imshow(im)


def show_annotation_and_image(image, annotation):
  '''
  Displays the image and its annotation side by side

  Args:
    image (numpy array) -- the input image
    annotation (numpy array) -- the label map
  '''
  new_ann = np.argmax(annotation, axis=2)
  seg_img = give_color_to_annotation(new_ann)

  image = image + 1
  image = image * 127.5
  image = np.uint8(image)
  images = [image, seg_img]

  images = [image, seg_img]
```

```
  images = [image, seg_img]
  fused_img = fuse_with_pil(images)
  plt.imshow(fused_img)


def list_show_annotation(dataset):
  '''
  Displays images and its annotations side by side

  Args:
    dataset (tf Dataset) - batch of images and annotations
  '''

  ds = dataset.unbatch()
  ds = ds.shuffle(buffer_size=100)

  plt.figure(figsize=(25, 15))
  plt.title("Images And Annotations")
  plt.subplots_adjust(bottom=0.1, top=0.9, hspace=0.05)

  # we set the number of image-annotation pairs to 9
  # feel free to make this a function parameter if you want
  for idx, (image, annotation) in enumerate(ds.take(9)):
    plt.subplot(3, 3, idx + 1)
    plt.yticks([])
    plt.xticks([])
    show_annotation_and_image(image.numpy(), annotation.numpy())
```

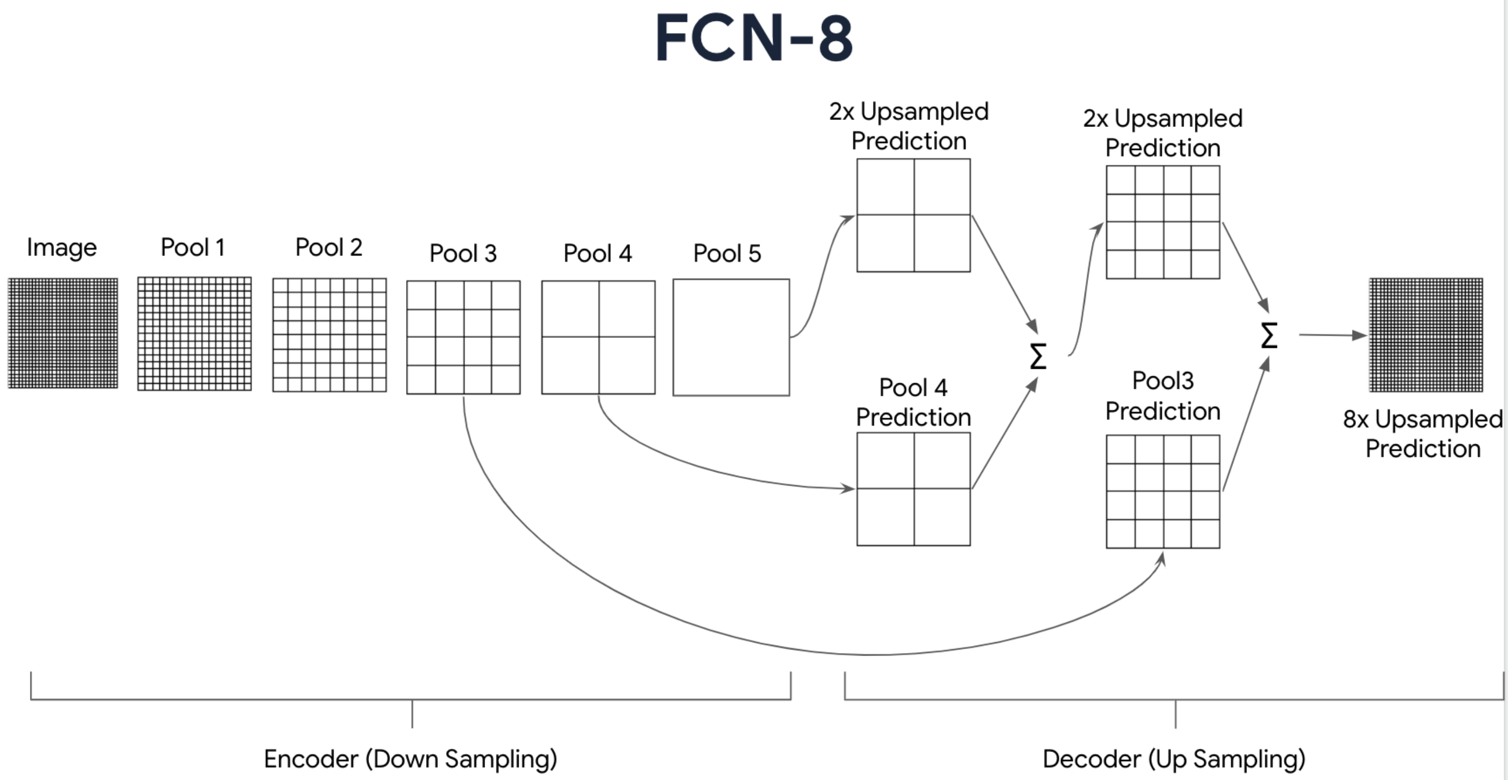
Please run the cells below to see sample images from the train and validation sets. You will see the image and the label maps side side by side.

```
list_show_annotation(training_dataset)
```

```
list_show_annotation(validation_dataset)
```

Define the Model

You will now build the model and prepare it for training. AS mentioned earlier, this will use a VGG-16 network for the encoder and FCN-8 for the decoder. This is the diagram as shown in class:



For this exercise, you will notice a slight difference from the lecture because the dataset images are 224x224 instead of 32x32. You'll see how this is handled in the next cells as you build the encoder.

Define Pooling Block of VGG

As you saw in Course 1 of this specialization, VGG networks have repeating blocks so to make the code neat, it's best to create a function to encapsulate this process. Each block has convolutional layers followed by a max pooling layer which downsamples the image.

```
def block(x, n_convs, filters, kernel_size, activation, pool_size, pool_stride, block_name):
  '''
  Defines a block in the VGG network.

  Args:
    x (tensor) -- input image
    n_convs (int) -- number of convolution layers to append
    filters (int) -- number of filters for the convolution layers
    activation (string or object) -- activation to use in the convolution
    pool_size (int) -- size of the pooling layer
    pool_stried (int) -- stride of the pooling layer
    block_name (string) -- name of the block

  Returns:
    tensor containing the max-pooled output of the convolutions
  '''
```

```
    for i in range(n_convs):
        x = tf.keras.layers.Conv2D(filters=filters, kernel_size=kernel_size, activation=activation, pa

    x = tf.keras.layers.MaxPooling2D(pool_size=pool_size, strides=pool_stride, name="{}_pool{}".format

    return x
```

Download VGG weights

First, please run the cell below to get pre-trained weights for VGG-16. You will load this in the next section when you build the encoder network.

```
# download the weights
!wget https://github.com/fchollet/deep-learning-models/releases/download/v0.1/vgg16_weights_tf_dim_o

# assign to a variable
vgg_weights_path = "vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5"
```

```
--2021-05-22 20:18:48--  https://github.com/fchollet/deep-learning-models/releases/download/v0
Resolving github.com (github.com)... 192.30.255.112
Connecting to github.com (github.com)|192.30.255.112|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://github-releases.githubusercontent.com/64878964/b09fedd4-5983-11e6-8f9f-904ea
--2021-05-22 20:18:49--  https://github-releases.githubusercontent.com/64878964/b09fedd4-5983-
Resolving github-releases.githubusercontent.com (github-releases.githubusercontent.com)... 185
Connecting to github-releases.githubusercontent.com (github-releases.githubusercontent.com)|18
HTTP request sent, awaiting response... 200 OK
Length: 58889256 (56M) [application/octet-stream]
Saving to: ‘vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5’

vgg16_weights_tf_di 100%[===================>]  56.16M  70.5MB/s    in 0.8s

2021-05-22 20:18:50 (70.5 MB/s) - ‘vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5’ saved [5
```

Define VGG-16

You can build the encoder as shown below.

- You will create 5 blocks with increasing number of filters at each stage.
- The number of convolutions, filters, kernel size, activation, pool size and pool stride will remain constant.
- You will load the pretrained weights after creating the VGG 16 network.
- Additional convolution layers will be appended to extract more features.
- The output will contain the output of the last layer and the previous four convolution blocks.

```
def VGG_16(image_input):
  '''
  This function defines the VGG encoder.

  Args:
    image_input (tensor) - batch of images
```

```
  Returns:
    tuple of tensors - output of all encoder blocks plus the final convolution layer
  '''

  # create 5 blocks with increasing filters at each stage.
  # you will save the output of each block (i.e. p1, p2, p3, p4, p5). "p" stands for the pooling lay
  x = block(image_input,n_convs=2, filters=64, kernel_size=(3,3), activation='relu',pool_size=(2,2)
  p1= x

  x = block(x,n_convs=2, filters=128, kernel_size=(3,3), activation='relu',pool_size=(2,2), pool_str
  p2 = x

  x = block(x,n_convs=3, filters=256, kernel_size=(3,3), activation='relu',pool_size=(2,2), pool_str
  p3 = x

  x = block(x,n_convs=3, filters=512, kernel_size=(3,3), activation='relu',pool_size=(2,2), pool_str
  p4 = x

  x = block(x,n_convs=3, filters=512, kernel_size=(3,3), activation='relu',pool_size=(2,2), pool_str
  p5 = x

  # create the vgg model
  vgg  = tf.keras.Model(image_input , p5)

  # load the pretrained weights you downloaded earlier
  vgg.load_weights(vgg_weights_path)

  # number of filters for the output convolutional layers
  n = 4096

  # our input images are 224x224 pixels so they will be downsampled to 7x7 after the pooling layers
  # we can extract more features by chaining two more convolution layers.
  c6 = tf.keras.layers.Conv2D( n , ( 7 , 7 ) , activation='relu' , padding='same', name="conv6")(p5
  c7 = tf.keras.layers.Conv2D( n , ( 1 , 1 ) , activation='relu' , padding='same', name="conv7")(c6

  # return the outputs at each stage. you will only need two of these in this particular exercise
  # but we included it all in case you want to experiment with other types of decoders.
  return (p1, p2, p3, p4, c7)
```

Define FCN 8 Decoder

Next, you will build the decoder using deconvolution layers. Please refer to the diagram for FCN-8 at the start of this section to visualize what the code below is doing. It will involve two summations before upsampling to the original image size and generating the predicted mask.

```
def fcn8_decoder(convs, n_classes):
  '''
  Defines the FCN 8 decoder.

  Args:
    convs (tuple of tensors) - output of the encoder network
    n_classes (int) - number of classes

  Returns:
    tensor with shape (height, width, n_classes) containing class probabilities
  '''
```

```
  # unpack the output of the encoder
  f1, f2, f3, f4, f5 = convs

  # upsample the output of the encoder then crop extra pixels that were introduced
  o = tf.keras.layers.Conv2DTranspose(n_classes , kernel_size=(4,4) ,  strides=(2,2) , use_bias=Fals
  o = tf.keras.layers.Cropping2D(cropping=(1,1))(o)

  # load the pool 4 prediction and do a 1x1 convolution to reshape it to the same shape of `o` above
  o2 = f4
  o2 = ( tf.keras.layers.Conv2D(n_classes , ( 1 , 1 ) , activation='relu' , padding='same'))(o2)

  # add the results of the upsampling and pool 4 prediction
  o = tf.keras.layers.Add()([o, o2])

  # upsample the resulting tensor of the operation you just did
  o = (tf.keras.layers.Conv2DTranspose( n_classes , kernel_size=(4,4) ,  strides=(2,2) , use_bias=Fa
  o = tf.keras.layers.Cropping2D(cropping=(1, 1))(o)

  # load the pool 3 prediction and do a 1x1 convolution to reshape it to the same shape of `o` above
  o2 = f3
  o2 = ( tf.keras.layers.Conv2D(n_classes , ( 1 , 1 ) , activation='relu' , padding='same'))(o2)

  # add the results of the upsampling and pool 3 prediction
  o = tf.keras.layers.Add()([o, o2])

  # upsample up to the size of the original image
  o = tf.keras.layers.Conv2DTranspose(n_classes , kernel_size=(8,8) ,  strides=(8,8) , use_bias=Fals

  # append a softmax to get the class probabilities
  o = (tf.keras.layers.Activation('softmax'))(o)

  return o
```

Define Final Model

You can now build the final model by connecting the encoder and decoder blocks.

```
def segmentation_model():
  '''
  Defines the final segmentation model by chaining together the encoder and decoder.

  Returns:
    keras Model that connects the encoder and decoder networks of the segmentation model
  '''

  inputs = tf.keras.layers.Input(shape=(224,224,3,))
  convs = VGG_16(image_input=inputs)
  outputs = fcn8_decoder(convs, 12)
  model = tf.keras.Model(inputs=inputs, outputs=outputs)

  return model
```

```
# instantiate the model and see how it looks
model = segmentation_model()
model.summary()
```

```
block2_conv2 (Conv2D)          (None, 112, 112, 128 147584      block2_conv1[0][0]

block2_pool2 (MaxPooling2D)    (None, 56, 56, 128)  0           block2_conv2[0][0]

block3_conv1 (Conv2D)          (None, 56, 56, 256)  295168      block2_pool2[0][0]

block3_conv2 (Conv2D)          (None, 56, 56, 256)  590080      block3_conv1[0][0]

block3_conv3 (Conv2D)          (None, 56, 56, 256)  590080      block3_conv2[0][0]

block3_pool3 (MaxPooling2D)    (None, 28, 28, 256)  0           block3_conv3[0][0]

block4_conv1 (Conv2D)          (None, 28, 28, 512)  1180160     block3_pool3[0][0]

block4_conv2 (Conv2D)          (None, 28, 28, 512)  2359808     block4_conv1[0][0]

block4_conv3 (Conv2D)          (None, 28, 28, 512)  2359808     block4_conv2[0][0]

block4_pool3 (MaxPooling2D)    (None, 14, 14, 512)  0           block4_conv3[0][0]

block5_conv1 (Conv2D)          (None, 14, 14, 512)  2359808     block4_pool3[0][0]

block5_conv2 (Conv2D)          (None, 14, 14, 512)  2359808     block5_conv1[0][0]

block5_conv3 (Conv2D)          (None, 14, 14, 512)  2359808     block5_conv2[0][0]

block5_pool3 (MaxPooling2D)    (None, 7, 7, 512)    0           block5_conv3[0][0]

conv6 (Conv2D)                 (None, 7, 7, 4096)   102764544   block5_pool3[0][0]

conv7 (Conv2D)                 (None, 7, 7, 4096)   16781312    conv6[0][0]

conv2d_transpose (Conv2DTranspo (None, 16, 16, 12)  786432      conv7[0][0]

cropping2d (Cropping2D)        (None, 14, 14, 12)   0           conv2d_transpose[0][0]

conv2d (Conv2D)                (None, 14, 14, 12)   6156        block4_pool3[0][0]

add (Add)                      (None, 14, 14, 12)   0           cropping2d[0][0]
                                                                conv2d[0][0]

conv2d_transpose_1 (Conv2DTrans (None, 30, 30, 12)  2304        add[0][0]

cropping2d_1 (Cropping2D)      (None, 28, 28, 12)   0           conv2d_transpose_1[0][0]

conv2d_1 (Conv2D)              (None, 28, 28, 12)   3084        block3_pool3[0][0]

add_1 (Add)                    (None, 28, 28, 12)   0           cropping2d_1[0][0]
                                                                conv2d_1[0][0]

conv2d_transpose_2 (Conv2DTrans (None, 224, 224, 12) 9216       add_1[0][0]

activation (Activation)        (None, 224, 224, 12) 0           conv2d_transpose_2[0][0]
==================================================================================================
Total params: 135,067,736
Trainable params: 135,067,736
Non-trainable params: 0
```

Compile the Model

Next, the model will be configured for training. You will need to specify the loss, optimizer and metrics. You will use `categorical_crossentropy` as the loss function since the label map is transformed to one hot encoded vectors for each pixel in the image (i.e. `1` in one slice and `0` for other slices as described earlier).

```
sgd = tf.keras.optimizers.SGD(lr=1E-2, momentum=0.9, nesterov=True)

model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])
```

Train the Model

The model can now be trained. This will take around 30 minutes to run and you will reach around 85% accuracy for both train and val sets.

```
# number of training images
train_count = 367

# number of validation images
validation_count = 101

EPOCHS = 170

steps_per_epoch = train_count//BATCH_SIZE
validation_steps = validation_count//BATCH_SIZE

history = model.fit(training_dataset,
                    steps_per_epoch=steps_per_epoch, validation_data=validation_dataset, validation_
```

```
Epoch 142/170
5/5 [==============================] - 9s 2s/step - loss: 0.5320 - accuracy: 0.8464 - val_l
Epoch 143/170
5/5 [==============================] - 9s 2s/step - loss: 0.5157 - accuracy: 0.8523 - val_l
Epoch 144/170
5/5 [==============================] - 9s 2s/step - loss: 0.5135 - accuracy: 0.8526 - val_l
Epoch 145/170
5/5 [==============================] - 10s 2s/step - loss: 0.5680 - accuracy: 0.8368 - val_
Epoch 146/170
5/5 [==============================] - 9s 2s/step - loss: 0.5267 - accuracy: 0.8482 - val_l
Epoch 147/170
5/5 [==============================] - 9s 2s/step - loss: 0.5209 - accuracy: 0.8510 - val_l
Epoch 148/170
5/5 [==============================] - 9s 2s/step - loss: 0.5321 - accuracy: 0.8447 - val_l
Epoch 149/170
5/5 [==============================] - 9s 2s/step - loss: 0.5350 - accuracy: 0.8455 - val_l
Epoch 150/170
5/5 [==============================] - 9s 2s/step - loss: 0.5158 - accuracy: 0.8521 - val_l
Epoch 151/170
5/5 [==============================] - 10s 2s/step - loss: 0.5091 - accuracy: 0.8544 - val_
Epoch 152/170
5/5 [==============================] - 9s 2s/step - loss: 0.5156 - accuracy: 0.8518 - val_l
Epoch 153/170
5/5 [==============================] - 9s 2s/step - loss: 0.5309 - accuracy: 0.8485 - val_l
Epoch 154/170
5/5 [==============================] - 9s 2s/step - loss: 0.5211 - accuracy: 0.8508 - val_l
Epoch 155/170
5/5 [==============================] - 9s 2s/step - loss: 0.5285 - accuracy: 0.8455 - val_l
```

```
Epoch 156/170
5/5 [==============================] - 9s 2s/step - loss: 0.5134 - accuracy: 0.8526 - val_l
Epoch 157/170
5/5 [==============================] - 10s 2s/step - loss: 0.5135 - accuracy: 0.8534 - val_
Epoch 158/170
5/5 [==============================] - 9s 2s/step - loss: 0.4971 - accuracy: 0.8591 - val_l
Epoch 159/170
5/5 [==============================] - 9s 2s/step - loss: 0.5059 - accuracy: 0.8540 - val_l
Epoch 160/170
5/5 [==============================] - 9s 2s/step - loss: 0.5046 - accuracy: 0.8566 - val_l
Epoch 161/170
5/5 [==============================] - 9s 2s/step - loss: 0.5203 - accuracy: 0.8484 - val_l
Epoch 162/170
5/5 [==============================] - 9s 2s/step - loss: 0.5148 - accuracy: 0.8524 - val_l
Epoch 163/170
5/5 [==============================] - 10s 2s/step - loss: 0.5078 - accuracy: 0.8550 - val_
Epoch 164/170
5/5 [==============================] - 9s 2s/step - loss: 0.5249 - accuracy: 0.8497 - val_l
Epoch 165/170
5/5 [==============================] - 9s 2s/step - loss: 0.5092 - accuracy: 0.8553 - val_l
Epoch 166/170
5/5 [==============================] - 9s 2s/step - loss: 0.4958 - accuracy: 0.8584 - val_l
Epoch 167/170
5/5 [==============================] - 9s 2s/step - loss: 0.4882 - accuracy: 0.8603 - val_l
Epoch 168/170
5/5 [==============================] - 9s 2s/step - loss: 0.4942 - accuracy: 0.8596 - val_l
Epoch 169/170
5/5 [==============================] - 10s 2s/step - loss: 0.5143 - accuracy: 0.8530 - val_
Epoch 170/170
5/5 [==============================] - 9s 2s/step - loss: 0.5036 - accuracy: 0.8572 - val_l
```

Evaluate the Model

After training, you will want to see how your model is doing on a test set. For segmentation models, you can use the intersection-over-union and the dice score as metrics to evaluate your model. You'll see how it is implemented in this section.

```
def get_images_and_segments_test_arrays():
  '''
  Gets a subsample of the val set as your test set

  Returns:
    Test set containing ground truth images and label maps
  '''
  y_true_segments = []
  y_true_images = []
  test_count = 64

  ds = validation_dataset.unbatch()
  ds = ds.batch(101)

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


  y_true_segments = y_true_segments[:test_count, : ,: , :]
  y_true_segments = np.argmax(y_true_segments, axis=3)
```

```
  return y_true_images, y_true_segments

# load the ground truth images and segmentation masks
y_true_images, y_true_segments = get_images_and_segments_test_arrays()
```

Make Predictions

You can get output segmentation masks by using the `predict()` method. As you may recall, the output of our segmentation model has the shape `(height, width, 12)` where `12` is the number of classes. Each pixel value in those 12 slices indicates the probability of that pixel belonging to that particular class. If you want to create the predicted label map, then you can get the `argmax()` of that axis. This is shown in the following cell.

```
# get the model prediction
results = model.predict(validation_dataset, steps=validation_steps)

# for each pixel, get the slice number which has the highest probability
results = np.argmax(results, axis=3)
```

Compute Metrics

The function below generates the IOU and dice score of the prediction and ground truth masks. From the lectures, it is given that:

$$IOU = \frac{area\_of\_overlap}{area\_of\_union}$$

$$DiceScore = 2 * \frac{area\_of\_overlap}{combined\_area}$$

The code below does that for you. A small smoothening factor is introduced in the denominators to prevent possible division by zero.

```
def compute_metrics(y_true, y_pred):
  '''
  Computes IOU and Dice Score.

  Args:
    y_true (tensor) - ground truth label map
    y_pred (tensor) - predicted label map
  '''

  class_wise_iou = []
  class_wise_dice_score = []

  smoothening_factor = 0.00001

  for i in range(12):
    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
```

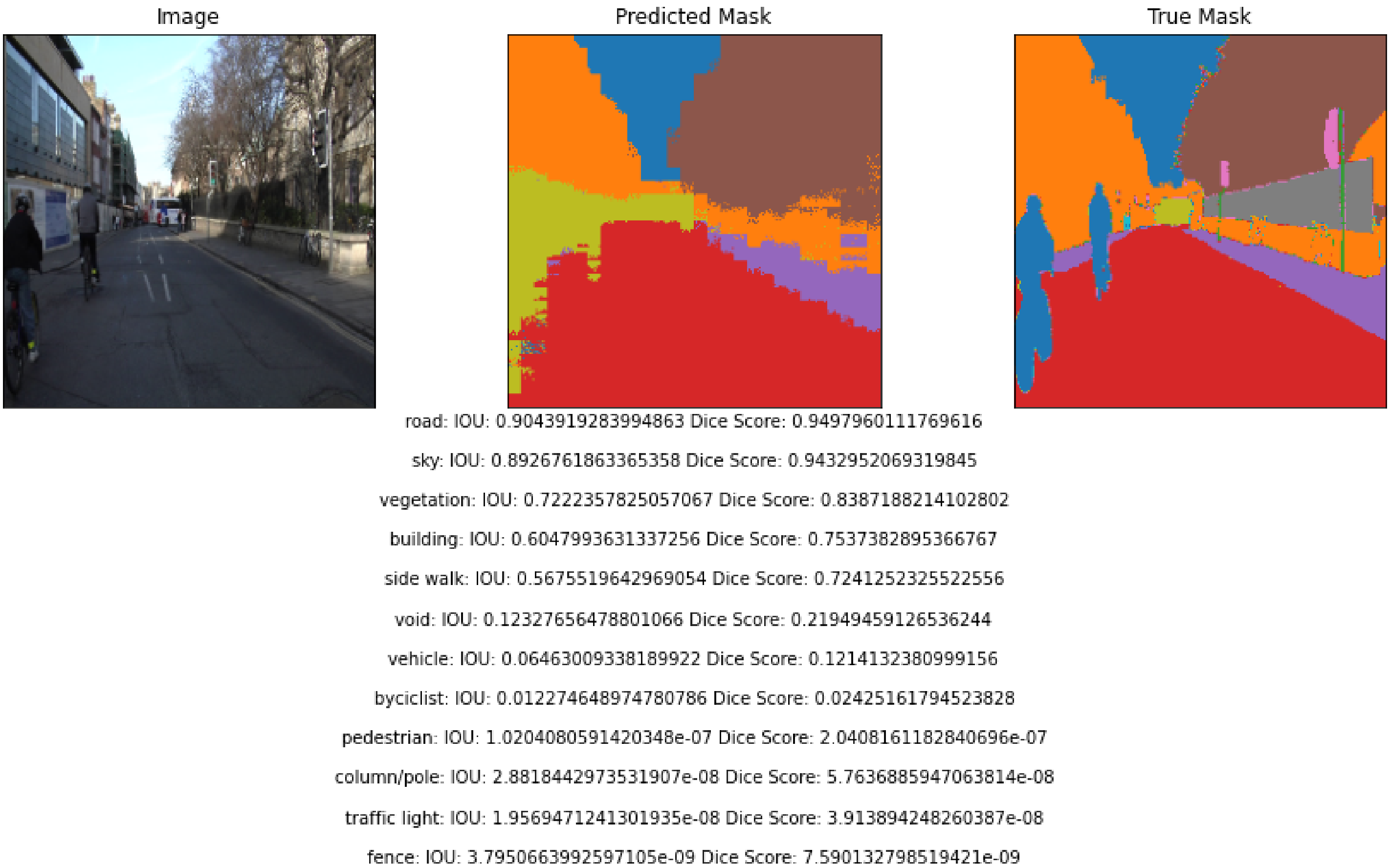
Show Predictions and Metrics

You can now see the predicted segmentation masks side by side with the ground truth. The metrics are also overlayed so you can evaluate how your model is doing.

```
# input a number from 0 to 63 to pick an image from the test set
integer_slider = 0

# compute metrics
iou, dice_score = compute_metrics(y_true_segments[integer_slider], results[integer_slider])

# visualize the output and metrics
show_predictions(y_true_images[integer_slider], [results[integer_slider], y_true_segments[integer_sl
```



Display Class Wise Metrics

You can also compute the class-wise metrics so you can see how your model performs across all images in the test set.

```
# compute class-wise metrics
cls_wise_iou, cls_wise_dice_score = compute_metrics(y_true_segments, results)
```

```
# print IOU for each class
for idx, iou in enumerate(cls_wise_iou):
  spaces = ' ' * (13-len(class_names[idx]) + 2)
  print("{}{}{} ".format(class_names[idx], spaces, iou))
```

```
sky            0.8911990444837634
building       0.7496353803396439
column/pole    4.6046875698279287e-10
road           0.9133212461447355
side walk      0.710456823117966
vegetation     0.8063245085748256
traffic light  2.949852506504468e-10
fence          0.00021196248263720107
vehicle        0.23725588919583787
pedestrian     4.038772211616005e-10
byciclist      0.0024589712379871013
void           0.15725934466288202
```

```
# print the dice score for each class
for idx, dice_score in enumerate(cls_wise_dice_score):
  spaces = ' ' * (13-len(class_names[idx]) + 2)
  print("{}{}{} ".format(class_names[idx], spaces, dice_score))
```

```
sky            0.942469854876563
building       0.8569046885631385
column/pole    9.209375139655857e-10
road           0.9546972292182997
side walk      0.8307217271182611
vegetation     0.8927792373453761
traffic light  5.899705013008936e-10
fence          0.0004238351281692416
vehicle        0.3835195148827734
pedestrian     8.07754442323201e-10
byciclist      0.004905879061143527
void           0.271778915245689
```

That's all for this lab! In the next section, you will work on another architecture for building a segmentation model: the UNET.

0s completed at 2:23 AM