Image Segmentation with U-Net

In this comprehensive article, we are going to build an U-Net, a type of CNN (Convolutional Neural Network) designed for quick, precise image segmentation, and use it to correctly predict a label for EVERY single pixel in the image. We are applying the model in images from a self-driving car dataset.

This type of image classification is called ***semantic image segmentation***. Like Object Detection, it answers the question "**WHAT** objects are in this image and **WHERE** in the image are those objects located?," but whereas Object Detection puts a bounding box over the object and may also include some pixels unrelated to that object, semantic image segmentation predicts a precise mask for each object in the image by labelling every single pixel with its corresponding class. The word “semantic” here refers to what's being shown, so for example the “Car” class is indicated below by the dark blue mask, and "Person" is indicated with a red mask:

As one may have guessed, region-specific labelling is absolutely crucial component of self-driving cars, which require a pixel-perfect understanding of their environment so they can change lanes and avoid other cars, or any number of traffic obstacles that can put peoples' lives in danger.

After reading this article, one can expect to:

- Build their own U-Net
- Distinguish between CNN and U-Net
- Implement semantic image segmentation on the CARLA self-driving car dataset

- Apply sparse categorical cross-entropy for pixel-wise prediction

---

Preprocessing

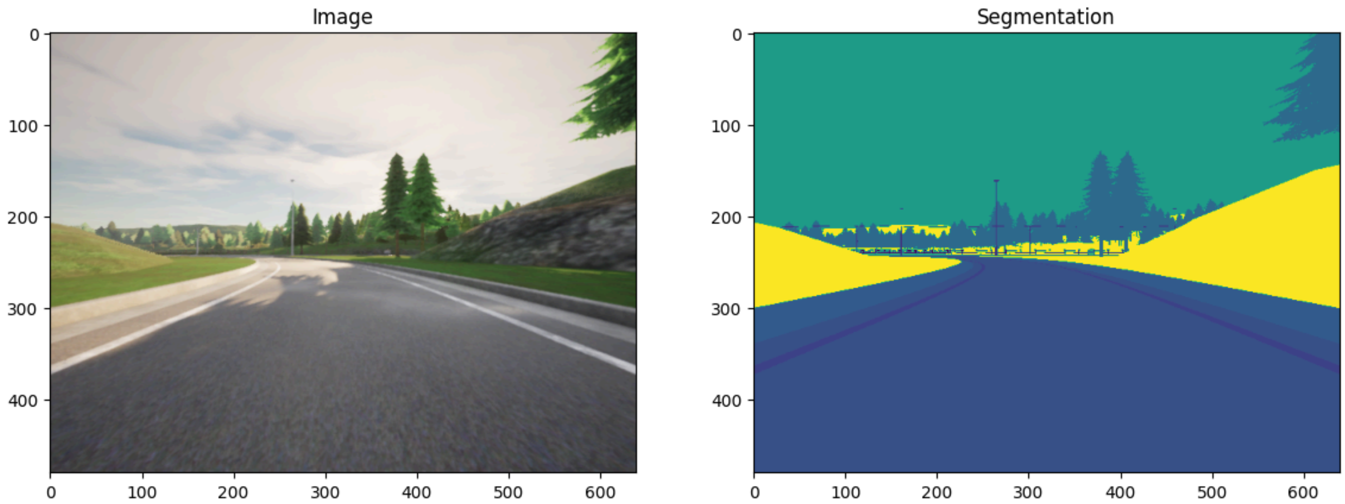
```
import tensorflow as tf
import numpy as np
import pandas as pd
import imageio
import matplotlib.pyplot as plt

%matplotlib inline

from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Conv2DTranspose
from tensorflow.keras.layers import concatenate
from tensorflow.keras.models import Model
```

```
path = ''

image_path = os.path.join(path, './data/CameraRGB/')
mask_path = os.path.join(path, './data/CameraMask/')
image_list_orig = os.listdir(image_path)
image_list = [image_path+i for i in image_list_orig]
mask_list = [mask_path+i for i in image_list_orig]
```

Let's check out some unmasked and masked images.

```
N = 35
img = imageio.imread(image_list[N])
mask = imageio.imread(mask_list[N])

fig, arr = plt.subplots(1, 2, figsize=(14, 10))
arr[0].imshow(img)
arr[0].set_title('Image')
```

```
arr[1].imshow(mask[:, :, 0])
arr[1].set_title('Segmentation')
```




```
image_list_ds = tf.data.Dataset.list_files(image_list, shuffle=False)
mask_list_ds = tf.data.Dataset.list_files(mask_list, shuffle=False)

image_filenames = tf.constant(image_list)
masks_filenames = tf.constant(mask_list)

dataset = tf.data.Dataset.from_tensor_slices((image_filenames,
masks_filenames))
```

We also normalize the images

```
def process_path(image_path, mask_path):
        img = tf.io.read_file(image_path)
        img = tf.image.decode_png(img, channels=3)
        img = tf.image.convert_image_dtype(img, tf.float32)

        mask = tf.io.read_file(mask_path)
        mask = tf.image.decode_png(mask, channels=3)
        mask = tf.math.reduce_max(mask, axis=-1, keepdims=True)

        return img, mask

def preprocess(image, mask):
        input_image = tf.image.resize(image, (96, 128),
method='nearest')
        input_mask = tf.image.resize(mask, (96, 128), method='nearest')
```

```
        return input_image, input_mask

image_ds = dataset.map(process_path)
processed_image_ds = image_ds.map(preprocess)
```

Theoretical Foundation

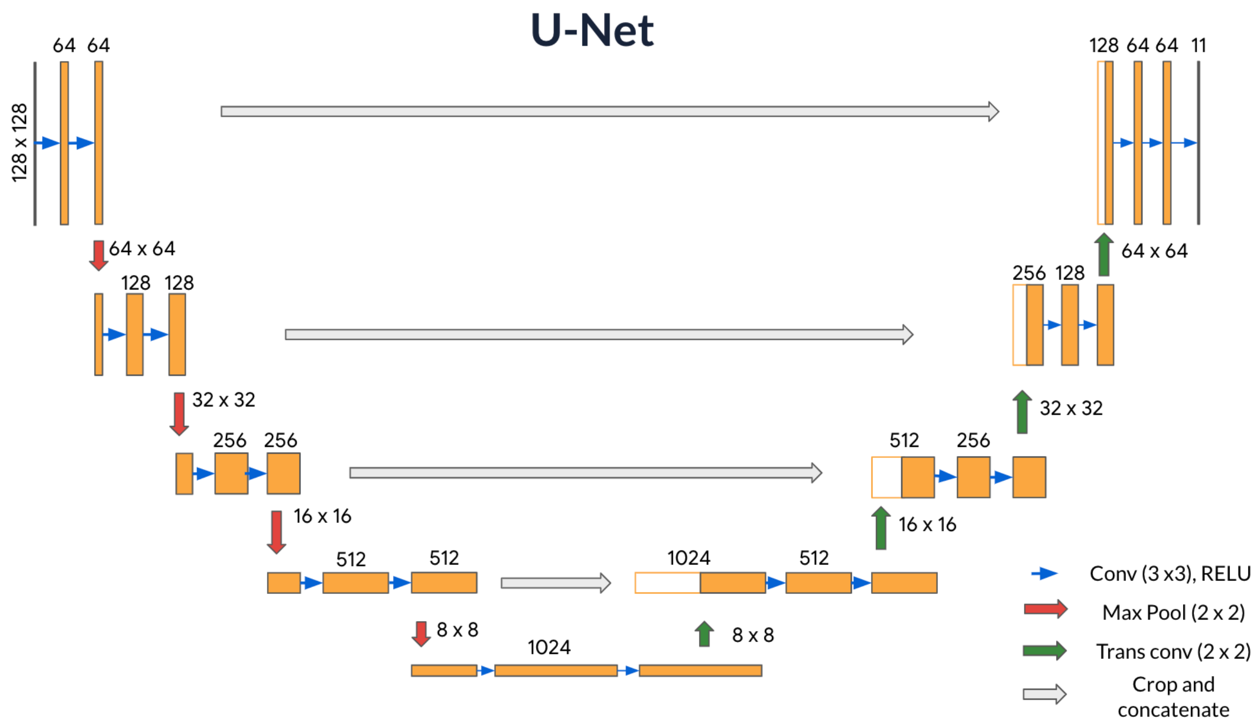
U-Net, named for its U-shape, was originally created in 2015 for tumor detection, but in years since then has become a very popular choice for other semantic image segmentation tasks.

U-Net builds upon a previous architecture called the Fully Convolutional Network (FCN), which replaces the dense layers found in a typical CMM with a ***transposed convolution layer*** that unsamples the feature map back to the size of the original input image, while preserving the spatial information. This preservation is essential since sampling through the dense layers destroy some spatial information(the 'where' of the image). It is also worth noticing that using transposed convolutions does not require the input size to be fixed, as it does when dense layers are used.

Unfortunately, the final feature layer of the FCN suffers from information loss due to downsampling too much. It then becomes difficult to unsample after so much information has been lost, causing an output to look rough.

This is when U-Net comes into play. Instead of one transposed convolution at the end of the network, it uses a matching number of convolutions for unsampling those feature maps back up to the *original input image size*. It also has ***skip connections***, to retain information that would have otherwise been lost during encoding. Skip connections send information to every unsampling layer in the decoder from the corresponding downsampling layer in the encoder, capturing finer information while also keeping the computation low. These help prevent information loss, as well as model overfitting.

Model Details



Contracting path (Encoder containing downsampling steps):

- Images are first fed through several convolutional layers which reduce height and width, while growing the number of channels.
- The contracting path follows a regular CNN architecture, with convolutional layers, their activations, and pooling layers to downsample the image and extract its features. In detail, it consists of the repeated application of two 3 x 3 same padding convolutions, each followed by a rectified linear unit (ReLU) and a 2 x 2 max pooling operation with stride 2 for downsampling. **At each downsampling step, the number of feature channels is doubled.**

Crop function: This step crops the image from the contracting path and concatenates it to the current image on the expanding path to create a skip connection.

Expanding path (Decoder containing upsampling steps):

- The expanding path performs the opposite operation of the contracting path, growing the image back to its original size, while shrinking the channels gradually.
- In detail, each step in the expanding path upsamples the feature map, followed by a 2 x 2 convolution (the transposed convolution). **This transposed convolution halves the number of feature channels, while growing the height and width of the image.**
- Next is a concatenation with the correspondingly cropped feature map from the contracting path, and two 3 x 3 convolutions, each followed by a ReLU. You need to

perform cropping to handle the loss of border pixels in every convolution.

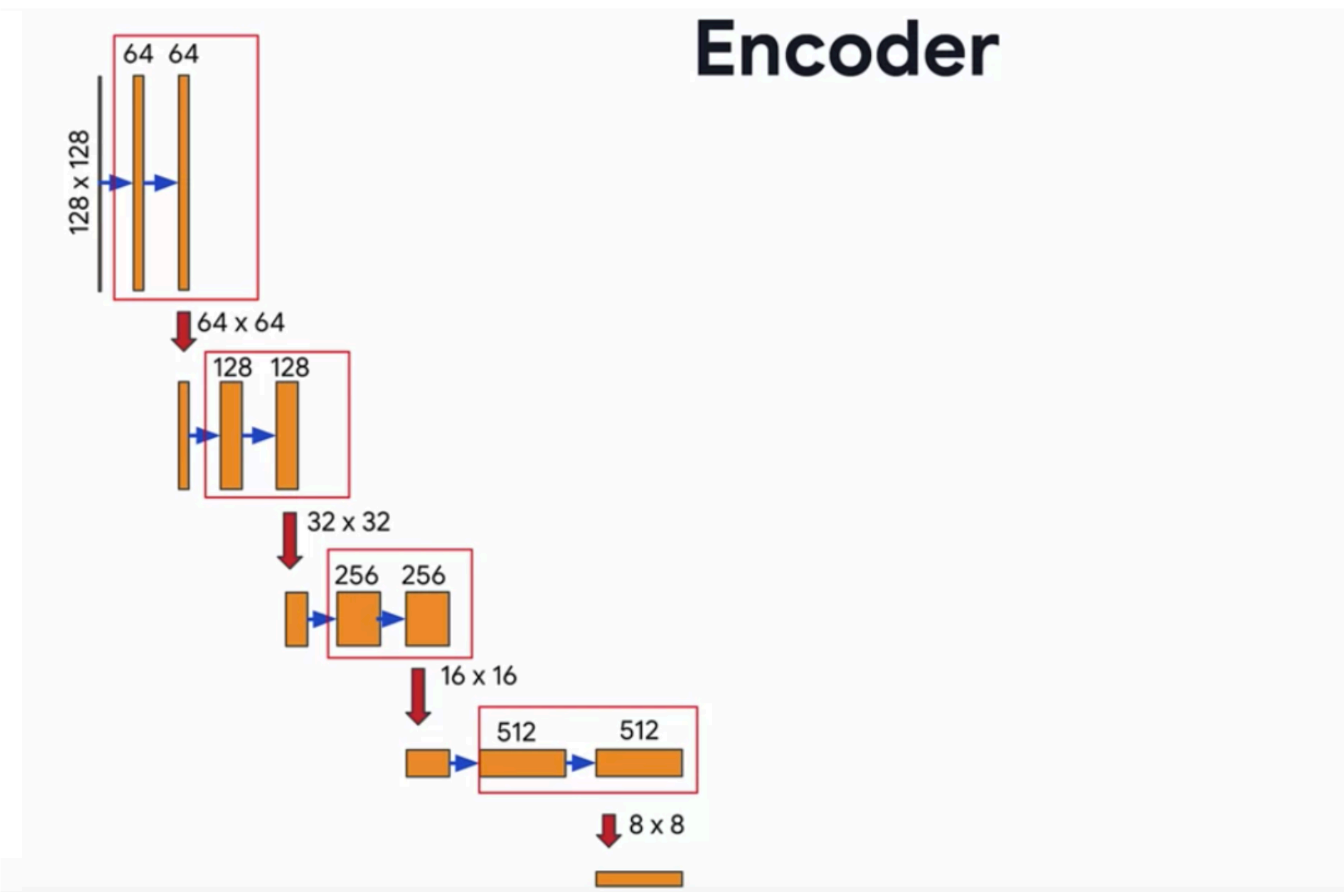
Final Feature Mapping Block: In the final layer, a 1x1 convolution is used to map each 64-component feature vector to the desired number of classes. The channel dimensions from the previous layer correspond to the number of filters used, so when you use 1x1 convolutions, you can transform that dimension by choosing an appropriate number of 1x1 filters. When this idea is applied to the last layer, you can reduce the channel dimensions to have one layer per class.

The U-Net network has 23 convolutional layers in total.

The important takeaway is that you multiply by 2 the number of filters used in the previous step.

Model Implementation

Encoder (Downsampling Block)




```
def conv_block(inputs=None, n_filters=32, dropout_prob=0,
max_pooling=True):
```

```
    """
    Convolutional downsampling block

    Arguments:
    inputs -- Input tensor
    n_filters -- Number of filters for the convolutional layers
    dropout_prob -- Dropout probability
    max_pooling -- Use MaxPooling2D to reduce the spatial dimensions
of the output volume

    Returns:
    next_layer, skip_connection -- Next layer and skip connection
outputs
    """

    conv = Conv2D(n_filters,
                  3,
                  activation='relu',
                  padding='same',
                  kernel_initializer='he_normal')(inputs)

    conv = Conv2D(n_filters, # Number of filters
                  3, # Kernel size
                  activation='relu',
                  padding='same',
                  kernel_initializer='he_normal')(conv)

    # if dropout_prob > 0 add a dropout layer, with the variable
dropout_prob as parameter
    if dropout_prob > 0:
        conv = Dropout(dropout_prob)(conv)

    # if max_pooling is True add a MaxPooling2D with 2x2 pool_size
    if max_pooling:
        next_layer = MaxPooling2D(pool_size=(2, 2))(conv)

    else:
        next_layer = conv
        skip_connection = conv

    return next_layer, skip_connection
```

Decoder (Upsampling block)

```
def upsampling_block(expansive_input, contractive_input, n_filters=32):
        """
        Convolutional upsampling block

        Arguments:
        expansive_input -- Input tensor from previous layer
        contractive_input -- Input tensor from previous skip layer
        n_filters -- Number of filters for the convolutional layers

        Returns:
        conv -- Tensor output

        """

        up = Conv2DTranspose(n_filters, # Number of filters
                                                  3, # Kernel size
                                                  strides=(2, 2),
                                                  padding='same')
(expansive_input)

        # Merge the previous output and the contractive_input
        merge = concatenate([up, contractive_input], axis=3)

        conv = Conv2D(n_filters, # Number of filters
                                  3, # Kernel size
                                  activation='relu',
                                  padding='same',
                                  kernel_initializer='he_normal')(merge)

        conv = Conv2D(n_filters, # Number of filters
                                  3, # Kernel size
                                  activation='relu',
                                  padding='same',
                                  kernel_initializer='he_normal')(conv)

        return conv
```

U-Net Model

We need to specify the number of output channels, which for this particular set would be 23. That's because there are 23 possible labels for each pixel in this self-driving car dataset.

For the function `unet_model`, specify the input shape, number of filters, and number of classes (23 in this case).

For the first half of the model:

- Begin with a conv block that takes the inputs of the model and the number of filters
- Then, chain the first output element of each block to the input of the next convolutional block
- Next, double the number of filters at each step
- Beginning with `conv_block4`, add `dropout_prob` of 0.3
- For the final conv_block, set `dropout_prob` to 0.3 again, and turn off max pooling

For the second half:

- Use cblock5 as expansive_input and cblock4 as contractive_input, with `n_filters` * 8. This is your bottleneck layer.
- Chain the output of the previous block as expansive_input and the corresponding contractive block output.
- Note that you must use the second element of the contractive block before the max pooling layer.
- At each step, use half the number of filters of the previous block
- `conv9` is a Conv2D layer with ReLU activation, He normal initializer, `same` padding
- Finally, `conv10` is a Conv2D that takes the number of classes as the filter, a kernel size of 1, and "same" padding. The output of `conv10` is the output of your model.

```
def unet_model(input_size=(96, 128, 3), n_filters=32, n_classes=23):
    """
    Unet model

    Arguments:
    input_size -- Input shape
    n_filters -- Number of filters for the convolutional layers
    n_classes -- Number of output classes

    Returns:
    model -- tf.keras.Model
```

```
        """

        inputs = Input(input_size)
        # Contracting Path (encoding)
        # Add a conv_block with the inputs of the unet_ model and
n_filters
        cblock1 = conv_block(inputs, n_filters)

        # Chain the first element of the output of each block to be the
input of the next conv_block.
        # Double the number of filters at each new step
        cblock2 = conv_block(cblock1[0], n_filters * 2)
        cblock3 = conv_block(cblock2[0], n_filters * 4)
        cblock4 = conv_block(cblock3[0], n_filters * 8,
dropout_prob=0.3)
        cblock5 = conv_block(cblock4[0], n_filters * 16,
dropout_prob=0.3, max_pooling=False)


        # Expanding Path (decoding)
        # Add the first upsampling_block.
        # Use the cblock5[0] as expansive_input and cblock4[1] as
contractive_input and n_filters * 8
        ublock6 = upsampling_block(cblock5[0], cblock4[1], n_filters *
8)

        # Chain the output of the previous block as expansive_input and
the corresponding contractive block output.
        # At each step, use half the number of filters of the previous
block
        ublock7 = upsampling_block(ublock6, cblock3[1], n_filters * 4)
        ublock8 = upsampling_block(ublock7, cblock2[1], n_filters * 2)
        ublock9 = upsampling_block(ublock8, cblock1[1], n_filters)

        conv9 = Conv2D(n_filters,
                       3,
                       activation='relu',
                       padding='same',
                       kernel_initializer='he_normal')
(ublock9)

        # Add a Conv2D layer with n_classes filter, kernel size of 1 and
a 'same' padding
```

```
        conv10 = Conv2D(n_classes, 1, padding='same')(conv9)

        model = tf.keras.Model(inputs=inputs, outputs=conv10)

        return model
```

Let's set the model dimension

```
img_height = 96
img_width = 128
num_channels = 3

unet = unet_model((img_height, img_width, num_channels))
```

and the summary:

```
unet.summary()
```

Layer (type)	Output Shape	Param #	Connecte
input_layer_23 (InputLayer)	(None, 96, 128, 3)	0	-
conv2d_84 (Conv2D)	(None, 96, 128, 32)	896	input_la
conv2d_85 (Conv2D)	(None, 96, 128, 32)	9,248	conv2d_8
max_pooling2d_29 (MaxPooling2D)	(None, 48, 64, 32)	0	conv2d_8
conv2d_86 (Conv2D)	(None, 48, 64, 64)	18,496	max_pool
conv2d_87 (Conv2D)	(None, 48, 64, 64)	36,928	conv2d_8
max_pooling2d_30 (MaxPooling2D)	(None, 24, 32, 64)	0	conv2d_8
conv2d_88 (Conv2D)	(None, 24, 32, 128)	73,856	max_pool

conv2d_89 (Conv2D)	(None, 24, 32, 128)	147,584	conv2d_8
max_pooling2d_31 (MaxPooling2D)	(None, 12, 16, 128)	0	conv2d_8
conv2d_90 (Conv2D)	(None, 12, 16, 256)	295,168	max_pool
conv2d_91 (Conv2D)	(None, 12, 16, 256)	590,080	conv2d_9
dropout_6 (Dropout)	(None, 12, 16, 256)	0	conv2d_9
max_pooling2d_32 (MaxPooling2D)	(None, 6, 8, 256)	0	dropout_
conv2d_92 (Conv2D)	(None, 6, 8, 512)	1,180,160	max_pool
conv2d_93 (Conv2D)	(None, 6, 8, 512)	2,359,808	conv2d_9
dropout_7 (Dropout)	(None, 6, 8, 512)	0	conv2d_9
conv2d_transpose_9 (Conv2DTranspose)	(None, 12, 16, 256)	1,179,904	dropout_
concatenate_9 (Concatenate)	(None, 12, 16, 512)	0	conv2d_t dropout_
conv2d_94 (Conv2D)	(None, 12, 16, 256)	1,179,904	concaten
conv2d_95 (Conv2D)	(None, 12, 16, 256)	590,080	conv2d_9
conv2d_transpose_10 (Conv2DTranspose)	(None, 24, 32, 128)	295,040	conv2d_9
concatenate_10 (Concatenate)	(None, 24, 32, 256)	0	conv2d_t conv2d_8
conv2d_96 (Conv2D)	(None, 24, 32, 128)	295,040	concaten
conv2d_97 (Conv2D)	(None, 24, 32, 128)	147,584	conv2d_9

conv2d_transpose_11 (Conv2DTranspose)	(None, 48, 64, 64)	73,792	conv2d_9
concatenate_11 (Concatenate)	(None, 48, 64, 128)	0	conv2d_t conv2d_8
conv2d_98 (Conv2D)	(None, 48, 64, 64)	73,792	concater
conv2d_99 (Conv2D)	(None, 48, 64, 64)	36,928	conv2d_9
conv2d_transpose_12 (Conv2DTranspose)	(None, 96, 128, 32)	18,464	conv2d_9
concatenate_12 (Concatenate)	(None, 96, 128, 64)	0	conv2d_t conv2d_8
conv2d_100 (Conv2D)	(None, 96, 128, 32)	18,464	concater
conv2d_101 (Conv2D)	(None, 96, 128, 32)	9,248	conv2d_1
conv2d_102 (Conv2D)	(None, 96, 128, 32)	9,248	conv2d_1
conv2d_103 (Conv2D)	(None, 96, 128, 23)	759	conv2d_1

Total params: 8,640,471 (32.96 MB)

Trainable params: 8,640,471 (32.96 MB)

Non-trainable params: 0 (0.00 B)

Loss Function

In semantic segmentation, you need as many masks as you have object classes. In the dataset you're using, each pixel in every mask has been assigned a single integer probability that it belongs to a certain class, from 0 to num_classes-1. The correct class is the layer with the highest probability.

This is different from *categorical crossentropy*, where the labels should be one-hot encoded (just 0s and 1s). Here, you'll use *sparse categorical crossentropy* as your loss function, to perform pixel-wise multiclass prediction. Sparse categorical crossentropy is more efficient than other loss functions when you're dealing with lots of classes.

```
unet.compile(optimizer='adam',
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
metrics=['accuracy'])
```

Training the Model

Firstly, let's create a function to display both an input image, and its ground truth: the true mask. The true mask is what your trained model output is aiming to get as close as possible.

```
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.preprocessing.image.array_to_img(display_list[i]))

                plt.axis('off')

                plt.show()
```

```
EPOCHS = 5
VAL_SUBSPLITS = 5
BUFFER_SIZE = 500
BATCH_SIZE = 32

train_dataset =
processed_image_ds.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
print(processed_image_ds.element_spec)
```

```
model_history = unet.fit(train_dataset, epochs=EPOCHS)
```

(TensorSpec(shape=(96, 128, 3), dtype=tf.float32, name=None), TensorSpec(shape=(96, 128, 1), dtype=tf.uint8, name=None))
Epoch 1/5
[1m34/34 [0m [32m━━━━━━━━━━━━━━━━━━━━ [0m [37m [0m
[1m102s [0m 3s/step - accuracy: 0.3855 - loss: 2.0897
Epoch 2/5
[1m34/34 [0m [32m━━━━━━━━━━━━━━━━━━━━ [0m [37m [0m
[1m111s [0m 3s/step - accuracy: 0.7938 - loss: 0.8400
Epoch 3/5
[1m34/34 [0m [32m━━━━━━━━━━━━━━━━━━━━ [0m [37m [0m
[1m113s [0m 3s/step - accuracy: 0.8327 - loss: 0.5903
Epoch 4/5
[1m34/34 [0m [32m━━━━━━━━━━━━━━━━━━━━ [0m [37m [0m
[1m111s [0m 3s/step - accuracy: 0.8639 - loss: 0.4720
Epoch 5/5
[1m34/34 [0m [32m━━━━━━━━━━━━━━━━━━━━ [0m [37m [0m
[1m116s [0m 3s/step - accuracy: 0.8924 - loss: 0.3573

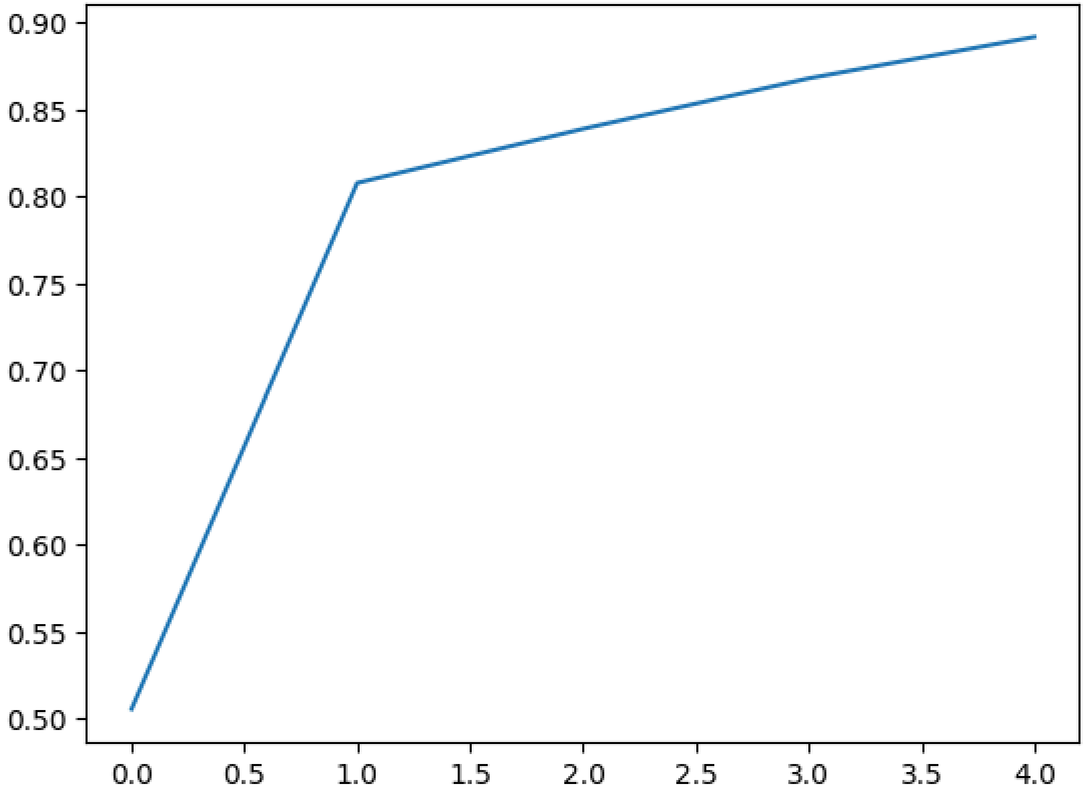
Creating Predicted Masks

The following function uses `tf.argmax` in the axis of the number of classes to return the index with the largest value and merge the prediction into a single image

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

        return pred_mask[0]
```

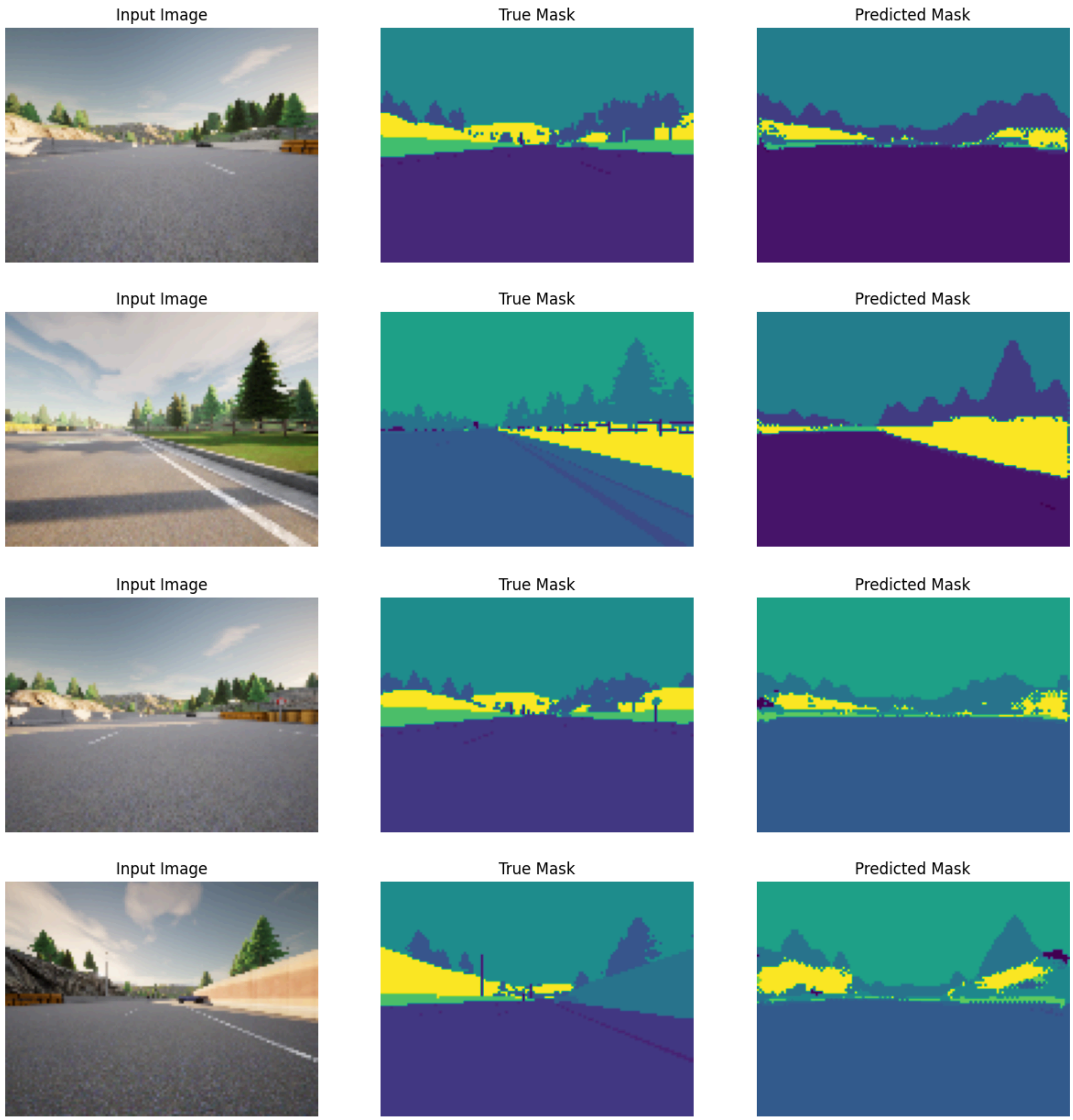
Let's plot the model accuracy

```
plt.plot(model_history.history["accuracy"])
```



Predictions

```
def show_predictions(dataset=None, num=1):
	"""
	Displays the first image of each of the num batches
	"""
	if dataset:
		for image, mask in dataset.take(num):
			pred_mask = unet.predict(image)
			display([image[0], mask[0], 
create_mask(pred_mask)])
	else:
		display([sample_image, sample_mask,

create_mask(unet.predict(sample_image[tf.newaxis, ...]))])

show_predictions(train_dataset, 4)
```



Training code to get consistent results:

```
EPOCHS = 15
VAL_SUBSPLITS = 5
BUFFER_SIZE = 500
BATCH_SIZE = 32


tf.keras.utils.set_random_seed(1)
```

```
tf.config.experimental.enable_op_determinism()

train_dataset =
processed_image_ds.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
print(processed_image_ds.element_spec)

unet = unet_model((img_height, img_width, num_channels))
unet.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                         metrics=['accuracy']

)

model_history = unet.fit(train_dataset, epochs=EPOCHS)
```