Follow Me Project

Introduction:

In this project, we applied semantic segmentation deep learning neural network technique to perform scene understanding task for assisting drone to follow specific target named hero. Semantic segmentation technique not only can provide the information whether the target present in the image, but also know where in the image the target is. This is the reason why the drone can automatically adjust its direction to follow the hero.

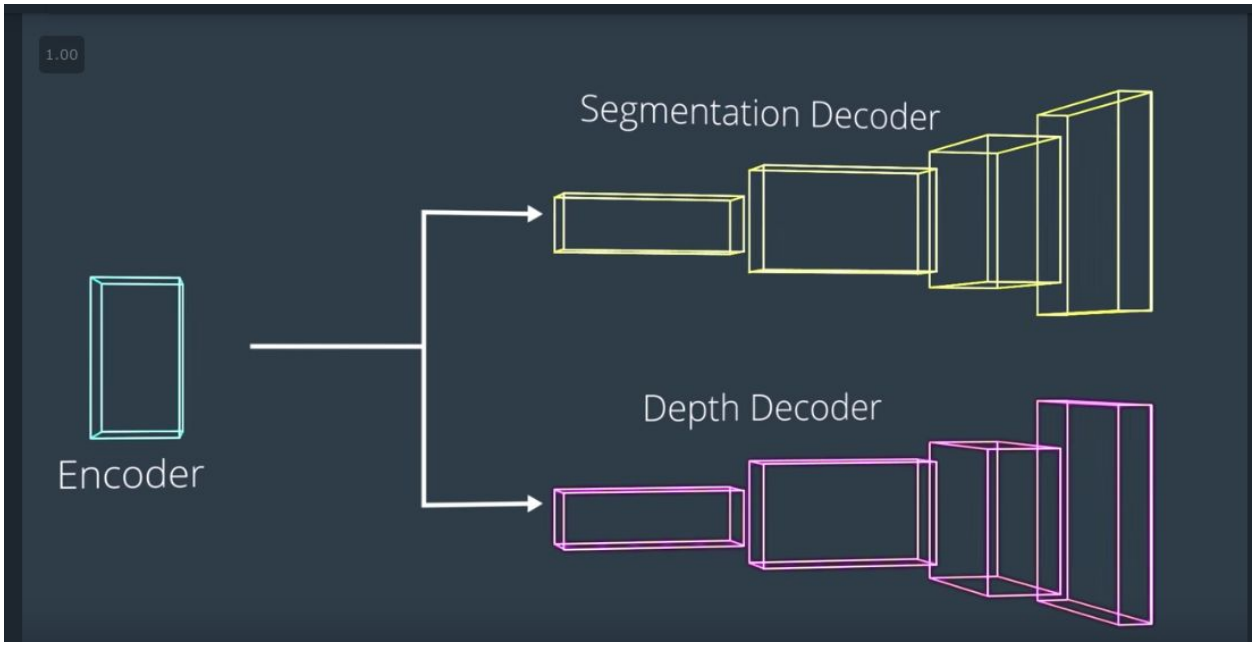


Figure 1: Architecture for scene understanding for autonomous vehicle.

Problem Overview:

Semantic segmentation gives the drone a capability to understand the true shape of target object. To achieve this, we used image data streaming from drone camera to train a fully convolutional neural network (FCN). Then, we apply the model to make predictions on the validation dataset images. The result will be compared with ground truth labels. Finally, we use intersection over union (IoU) to give scores to evaluate how good the model is. To build FCN, we will follow the architecture in Figure 2.

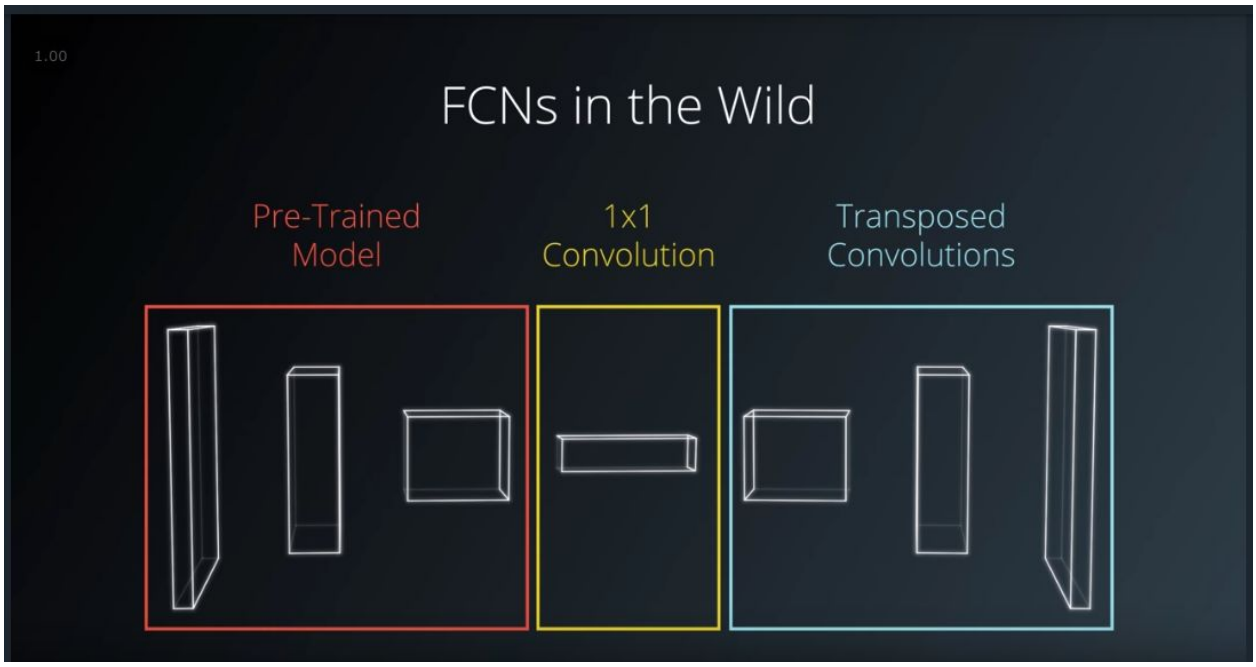


Figure 2: FCN architecture.

FCN architecture:

Figure 3 shows the FCN architecture to solve the drone following target problem.

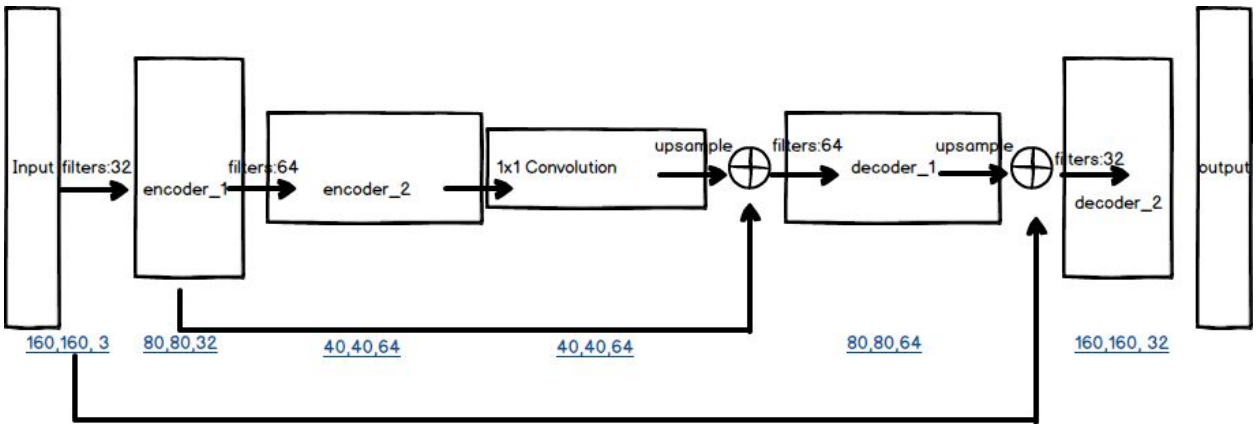


Figure 3: Programming the fully convolutional neural network

Overview of FCN:

The FCN consists of two encoder layers, one 1x1 convolution layer and 2 decoder layers. The encoder for the FCN implements separable convolution layers with batch normalization and with the ReLU activation function. The separable convolution technique tremendously reduce the number of parameters and also reduce overfitting to an extent. Batch normalization normalize

the inputs to layers within the network, which allows us to train networks fasters, allows to use higher learning rates, simplifies the creation of deeper network and provides regularization.

Encoder Layer:

The purpose of the two encoder layers is to extract features from the image dataset what will later feed into decoder layer. The weights and biases we learn for a given output layer are shared across all patches in a given input layer.

1x1 Convolution layer:

1x1 convolution layer is a very important layer. If we try to feed the output of a separable convolution layer into a fully connected layer, we flatten it into a 2D tensor. This will loss of spatial information. If we want let the drone know the target location on the image, we have to use 1x1 convolution layer to preserve target spatial information. Also, 1x1 convolution layer helps in reducing the dimensionality of the layer

The code below shows separable convolutions and encoder blocks

```
def separable_conv2d_batchnorm(input_layer, filters, strides=1):
    output_layer = SeparableConv2DKeras(filters=filters,kernel_size=3,
strides=strides, padding='same', activation='relu')(input_layer)
    output_layer = layers.BatchNormalization()(output_layer)
    return output_layer
```

```
def conv2d_batchnorm(input_layer, filters, kernel_size=3, strides=1):
    output_layer = layers.Conv2D(filters=filters, kernel_size=kernel_size,
strides=strides,
                      padding='same', activation='relu')(input_layer)

    output_layer = layers.BatchNormalization()(output_layer)
    return output_layer
```

```
def encoder_block(input_layer, filters, strides):
    # TODO Create a separable convolution layer using the
separable_conv2d_batchnorm() function.
    output_layer = separable_conv2d_batchnorm(input_layer, filters,
strides)
    return output_layer
```

Encoders:

The decoder block is comprised three parts. First, A bilinear upsampling layer using the upsample_bilinear() function. That's why we can concatenate small input layer with large input layer later. The bilinear upsampling is a resampling technique that utilizes the weighted average of four nearest known pixels, located diagonally to a given pixel, to estimate a new pixel intensity value.The bilinear upsampling method does not contribute as a learnable layer like the transposed convolutions in the architecture and is prone to lose some finer details, but it helps speed up performance. Second, skip connection which concatenate a upsampled small input layer to a non adjacent larger encoded layer . Third, we use additional separable convolution layers to extract some spatial information. The code below shows the implementation of decoder layer.

```
def fcn_model(inputs, num_classes):

    # TODO Add Encoder Blocks.
    # Remember that with each encoder layer, the depth of your model (the
number of filters) increases.
    encoder_layer_1 = encoder_block(inputs, filters=32, strides=2);
    print("encoder_layer_1 shape is ", encoder_layer_1.shape);
    encoder_layer_2 = encoder_block(encoder_layer_1, filters=64,
strides=2);
    print("encoder_layer_2 shape is ", encoder_layer_2.shape)
    # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
    convolution_1D_layer_1 = conv2d_batchnorm(encoder_layer_2, filters=32,
kernel_size=1, strides=1)
    print("convolution_1D_layer_1 shape is", convolution_1D_layer_1.shape)
    # TODO: Add the same number of Decoder Blocks as the number of Encoder
Blocks
    x_1 = decoder_block(convolution_1D_layer_1, encoder_layer_1,
filters=64)
    print("x_1:", x_1.shape)
    x = decoder_block(x_1, inputs, filters=32)
    print("x:", x.shape)
    # The function returns the output layer of your model. "x" is the final
layer obtained from the last decoder_block()
    return layers.Conv2D(num_classes, 1, activation='softmax',
padding='same')(x)
```

Hyper Parameters:

Learning_rate:

Lowing the learning rate can increase the model performance.

Batch_size:

batching is a technique for training on subsets of the dataset instead of all the data at one time. The batch size is the number of training samples/images that get propagated through the network in a single pass.

Num_epochs:

number of times the entire training dataset gets propagated through the network. An epoch is a single forward and backward pass of the whole dataset. This is used to increase the accuracy of the model without requiring more data. Lowering the learning rate would require more epochs, but could ultimately achieve better accuracy. I can see from my test, when I decrease the learning rate, but keep the num_epochs the same, the result got worse.

Steps_per_epoch:

number of batches of training images that go through the network in 1 epoch. We have provided you with a default value. One recommended value to try would be based on the total number of images in training dataset divided by the batch_size.

Validation_steps:

number of batches of validation images that go through the network in 1 epoch. This is similar to steps_per_epoch, except validation_steps is for the validation dataset. We have provided you with a default value for this as well.

Workers:

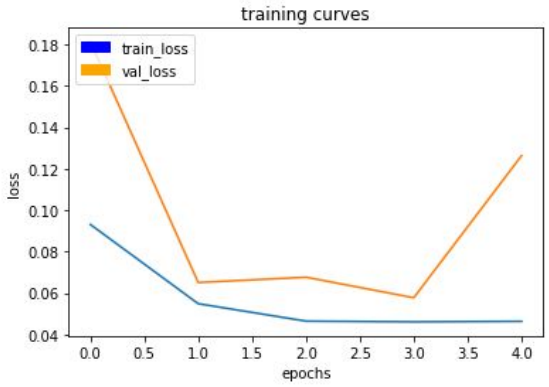
maximum number of processes to spin up. It depends the hardware you are using.

Experiments:

Run_1: hyper parameters

```
learning_rate = 0.05
batch_size = 20
num_epochs = 5
steps_per_epoch = 200
validation_steps = 50
workers = 2
```

Run_1: results



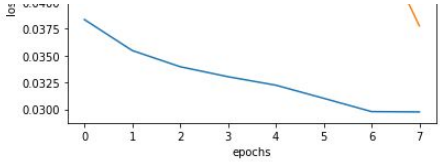

```
200/200 [==============================] - 924s - loss: 0.0463 - val_loss: 0.1263
```

Run_1 didn't get above 40% for the final score. Therefore, I decrease the learning rate, increase the batch_size, increase the num_epochs. The total images for training dataset is 4123. The total images for validation dataset is about 1823. If I increase the batch_size, I should decrease the steps_per_epoch and validation_steps. That's what happend in run_2

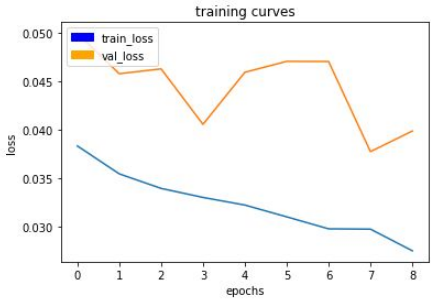
Run_2: hyper parameters

```
]: learning_rate = 0.02
   batch_size = 100
   num_epochs = 10
   steps_per_epoch = 40
   validation_steps = 20
   workers = 2
```

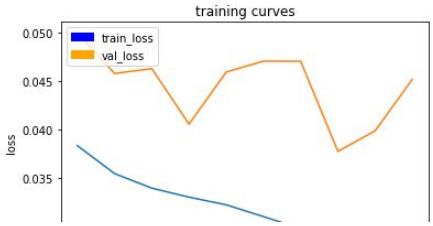
Run_2: Result




```
40/40 [==============================] - 995s - loss: 0.0301 - val_loss: 0.0378
Epoch 9/10
39/40 [==============================>.] - ETA: 21s - loss: 0.0276
```




```
40/40 [==============================] - 987s - loss: 0.0276 - val_loss: 0.0399
Epoch 10/10
39/40 [==============================>.] - ETA: 21s - loss: 0.0284
```



Run_2 score:

In [34]:
```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)
```

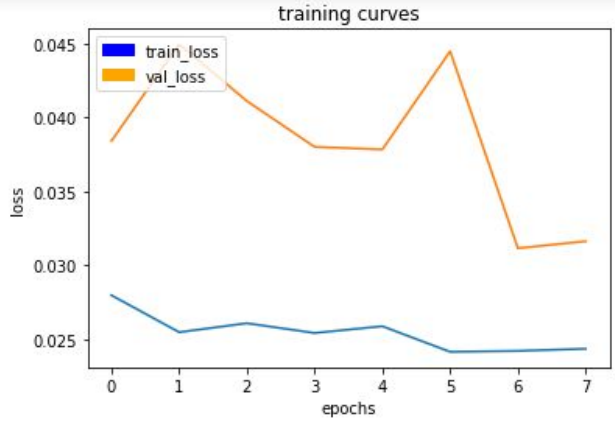
```
0.37589773390391
```

Run_2 score is better than run_1. Then, I tried to increase the batch_size again, decrease the validation_steps because validation loss is not very good.

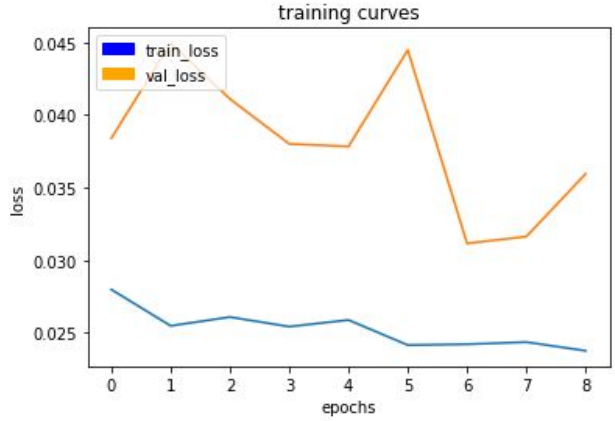
Run_3: hyper parameters,

In [35]:
```
learning_rate = 0.02
batch_size = 120
num_epochs = 10
steps_per_epoch = 40
validation_steps = 12
workers = 2
```

Run_3: score




```
40/40 [==============================] - 1151s - loss: 0.0244 - val_loss: 0.0316
Epoch 9/10
39/40 [==============================>.] - ETA: 26s - loss: 0.0238
```




```
40/40 [==============================] - 1144s - loss: 0.0237 - val_loss: 0.0360
Epoch 10/10
39/40 [==============================>.] - ETA: 26s - loss: 0.0233
```

In [48]:
```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)
```

```
0.42183560682890775
```

Future Enhancements:

I should try to build three layers of encoder and decoder with filter sizes 32, 64, 128. It would make the model detect finer details of the target. I also should increase the epochs because lowering learning rate corresponding with increasing epochs. I believe the model and would work well for following another object, like dog, cat or car because semantic segmentation is understanding an image at pixel level, which means that it assigns each pixel in the image an object class. That's why we use 1x1 convolutions to extract features from images and use a fully connected layer to classify these features.

Reference:
https://stats.stackexchange.com/questions/194142/what-does-1x1-convolution-mean-in-a-neural-network
https://stats.stackexchange.com/questions/182102/what-do-the-fully-connected-layers-do-in-cnns
http://blog.qure.ai/notes/semantic-segmentation-deep-learning-review