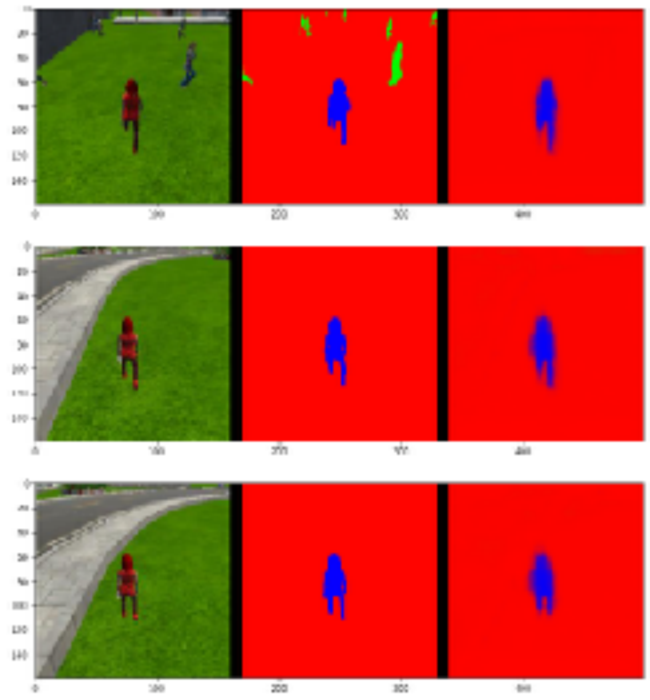
Introduction

Using a Fully Convolutional Network (FCN) from the Keras library, the Quad-Copter was able to identify the target person (red hair, red shirt) from an image. As seen from the video, the FCN was able to distinguish the target from the rest of the humans (Target in blue, other humans in green and background in pink).

After the target is identified, the Quad-Copter would now follow the target (as seen from the Blue region in the middle). It is important to note that this model does not work well to follow another object such as a cat/dog as we are only training it with classified images of people (red for background, blue for target human and green for other humans). The model would only work to follow other objects if there are training data of these images fed into the model.

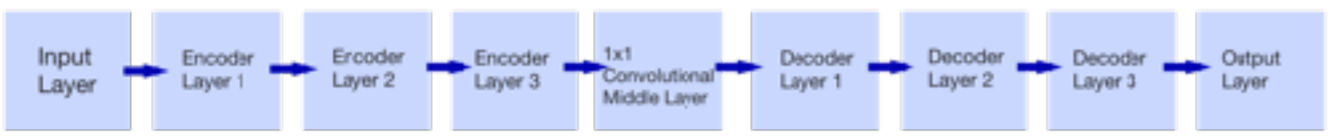


Network Architecture

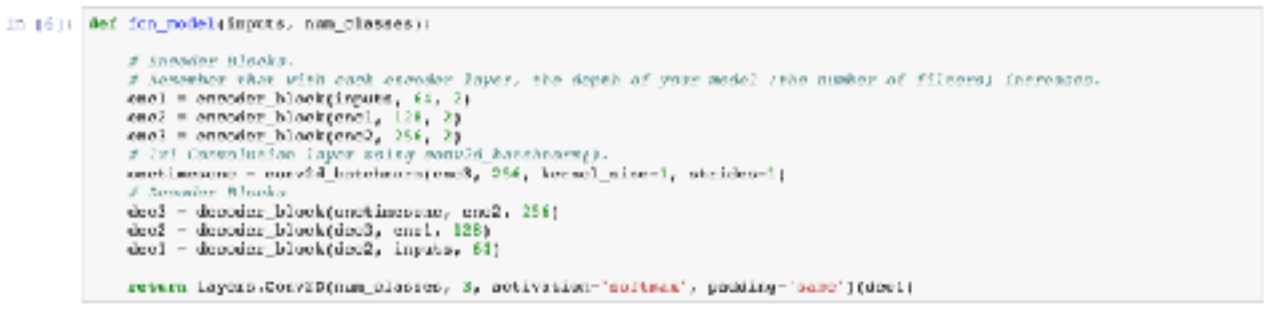
A fully convolutional network (FCN) was used to train the semantic segmentation model. The convolutional neural network contains 3 encoder blocks, followed by 1x1 convolution middle layer and 3 decoder blocks.

The encoder is used to reduce the spatial dimension of the image using the pooling technique before passing to the 1x1 convolution middle layer. The encoder is a series of convolutional layer where a sling window slides through the image. This is to extract features from the image and translate them to a more abstract representation.The convolution layer is to preserve spatial information. The decoder then recovers the spatial dimension for pixel-wise classification by upsampling the encoded image to have the same size as the input image.

Each decoder block has 1 bilinear upsampling layer, a layer concatenation step and 1 separable convolution layer.

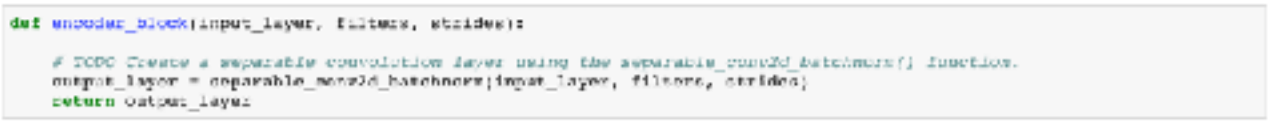


FCN Model Code



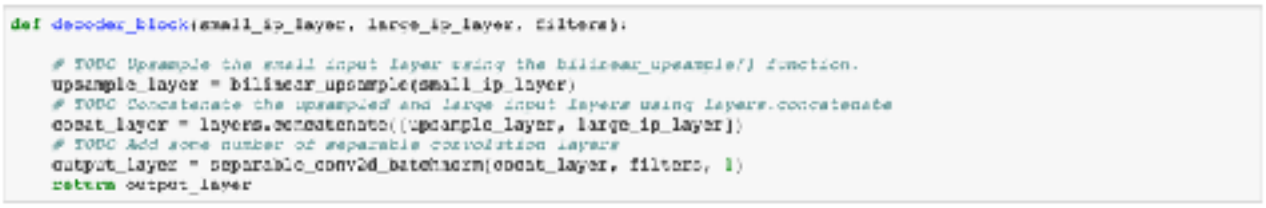
```
In [6]: def fcn_model(inputs, num_classes):

            # Encoder Blocks.
            # Remember that with each encoder layer, the depth of your model (the number of filters) increases.
            enc1 = encoder_block(inputs, 64, 2)
            enc2 = encoder_block(enc1, 128, 2)
            enc3 = encoder_block(enc2, 256, 2)
            # 1x1 Convolution layer using conv2d_batchnorm().
            onebyoneconv = conv2d_batchnorm(enc3, 256, kernel_size=1, strides=1)
            # Decoder Blocks
            dec3 = decoder_block(onebyoneconv, enc2, 256)
            dec2 = decoder_block(dec3, enc1, 128)
            dec1 = decoder_block(dec2, inputs, 64)

            return layers.Conv2D(num_classes, 3, activation='softmax', padding='same')(dec1)
```

Encoder Block



```
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
```

Decoder Block



```
def decoder_block(small_ip_layer, large_ip_layer, filters):

    # TODO Upsample the small input layer using the bilinear_upsample() function.
    upsample_layer = bilinear_upsample(small_ip_layer)
    # TODO Concatenate the upsampled and large input layers using layers.concatenate
    concat_layer = layers.concatenate([upsample_layer, large_ip_layer])
    # TODO Add some number of separable convolution layers
    output_layer = separable_conv2d_batchnorm(concat_layer, filters, 1)
    return output_layer
```

Choosing various parameters

Learning Rates
The learning rate is a value that sets how quickly (or slowly) a neural network makes adjustments to what it has learned while it is being trained. Learning rates that are too large would cause error rates to increase instead of decreasing. Learnings rates that are too small would result in overfitting. A small learning rate would also .. in a lot more time needed.

Batch Size
Number of images that are processed together as one batch for every epoch step. Good batch sizes should not be too large to cause overfitting and should be large enough to allow the available computing resources to handle it.

Epochs
Single complete pass over all of the available training data. A good epoch value is the lowest possible value that would allow the error rates of the neural network to be minimised.

Steps per Epoch
Number of batches of training images that go through the network during a single epoch. It is basically the number of images used by the neural network each time it computes and updates the gradient and error.

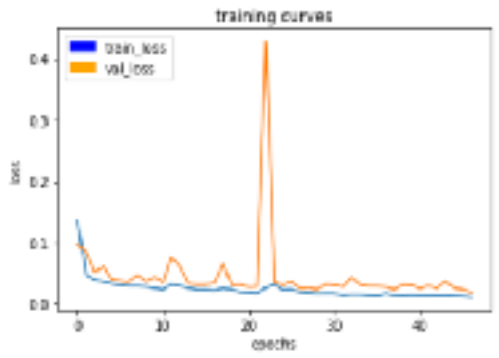
Validation Steps
Steps per Epoch for validation data.

Workers
Workers are the number of instances to spin up. Larger values tended to significantly slow down the initiation of the gradient descent computation.

Experiments

Experiment 1 — Using Udacity Dataset

Learning Rate = 0.01
Batch_size = 32
Num_epochs = 50
Steps per epoch = 200 (default)
Validation Steps = 50
Workers = 2



```
200/200 [==============================] - 150s - loss: 0.0117 - val_loss: 0.0191
```

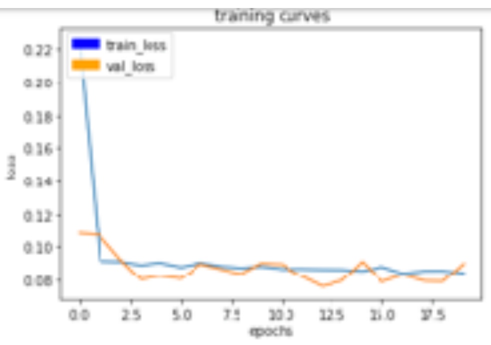
```
In [21]: # And the final grade score is
         final_score = final_IoU * weight
         print(final_score)

0.201301345610
```

Final Score : 0.201

With the low score, I tried to reduce the learning rate and I also reduced the number of epochs to 20 as it seems like it does not have much improvements after the 20th epoch. I added the to the training data by flipping the images (according to a discussion thread on slack) to increase the number of training images and masks fed to the FCN.

Experiment 2 — Using Udacity Dataset and flipped images

Learning Rate = 0.002
Batch_size = 32
Num_epochs = 20
Steps per epoch = 200
Validation Steps = 30
Workers = 4



```
200/200 [==============================] - 149s - loss: 0.0944 - val_loss: 0.0900
```

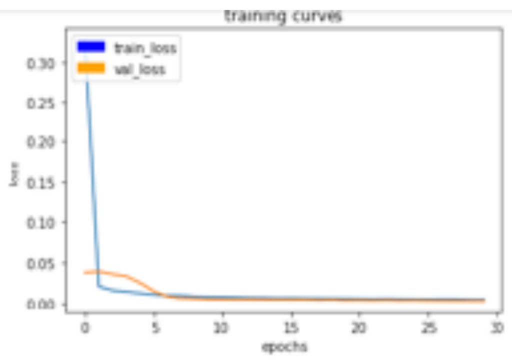
```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)

0.254976102339
```

Final Score : 0.254

I added my own training data to include more pictures of the target - target that is far away.

Experiment 3 — Using own images

Learning Rate = 0.002
Batch_size = 16
Num_epochs = 30
Steps per epoch = 100
Validation Steps = 50
Workers = 4



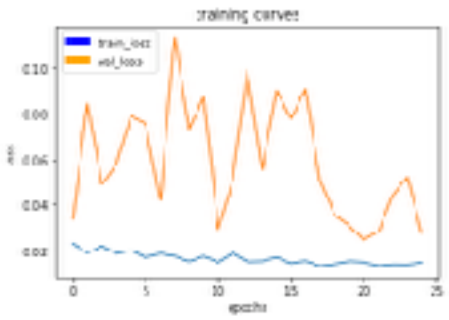
```
100/100 [==============================] - 40s - loss: 0.0044 - val_loss: 0.0030
```

```
In [21]: # And the final grade score is
         final_score = final_IoU * weight
         print(final_score)
0.268399182 71
```

Final Score : 0.268

Experiment 4 — Using own images + Udacity dataset

Learning Rate = 0.001
Batch_size = 16
Num_epochs = 25
Steps per epoch = 200
Validation Steps = 50
Workers = 4



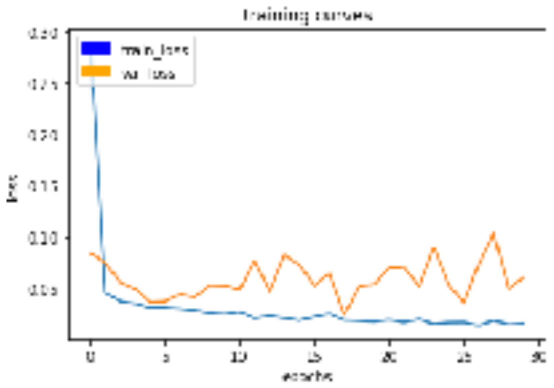
```
200/200 [==============================] - 77s - loss: 0.0149 - val_loss: 0.0277
```

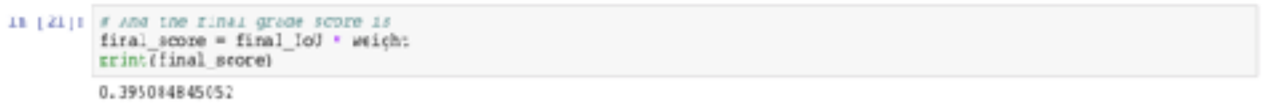
```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)
0.356434637941
```

Final Score: 0.356

Experiment 5 — Using own images + Udacity dataset

Learning Rate = 0.001
Batch_size = 16
Num_epochs = 30
Steps per epoch = 200
Validation Steps = 50
Workers = 4





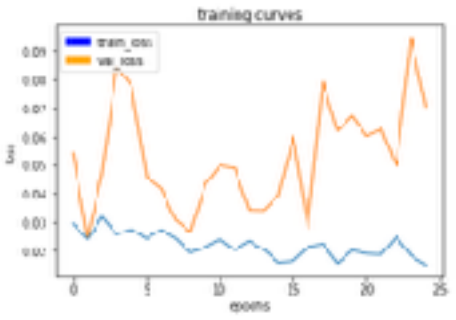
```
In [21]: # And the final grade score is
         final_score = final_IoU * weight
         print(final_score)

         0.395084845052
```

Final Score: 0.395 (nearest 2 dp)

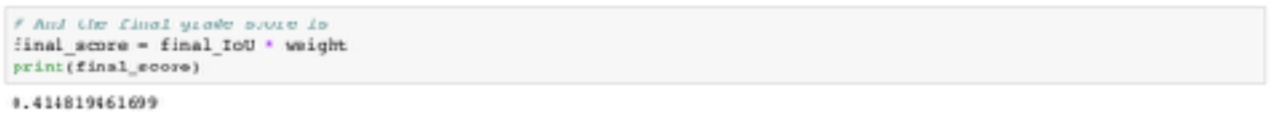
To reduce the possibility of overfitting, I increased the learning rate to 0.005 and also reduced the steps per epoch. I managed to finally have my accuracy above 0.40.

Experiment 6 — Using own images + Udacity dataset

Learning Rate = 0.005
Batch_size = 16
Num_epochs = 25
Steps per epoch = 100
Validation Steps = 50
Workers = 4



```
100/100 [==============================] - 41s - loss: 0.0152 - val_loss: 0.0706
```



```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)

0.414819461609
```

Final Score: 0.415

Discussion

As we can see from the above, the training dataset is highly important to ensure model performance. In this project, I had added a lot more photos that show target far away so that the model would be able to learn that. To speed up the time needed to train the model, I used batch size 16 instead of 32.

Simulation

The model files used are model_weights_given_set.h5 and config_model_weights_given_set.h5

Further Improvements

I would add more data images to the training and validation data sets to improve the learning and prevent the neural network from overfitting to the data I've fed. I would also increase the number of layers (increase the number of encoders and decoders used). Another possible way to improve is to increase the amount of data which contains the target and where the target is far away.

Resources used:

https://medium.com/towards-data-science/image-augmentation-for-deep-learning-histogram-equalization-a71387f609b2
https://stats.stackexchange.com/questions/164876/tradeoff-batch-size-vs-number-of-iterations-to-train-a-neural-network/236393
http://cv-tricks.com/image-segmentation/transpose-convolution-in-tensorflow/
https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html