Network Architecture

The FCN is built to be able to segment objects within video stream. This means each pixel in the image needs to be labeled. Fully convolutional networks are able to perform this in a process called semantic segmentation. The output image is the same size as the original input image, and each pixel in the output image is coloured one of N segmentation colours.
Semantic segmentation allows FCNs to preserve spatial information throughout the network.

Fully Convolutional Network:

A Fully Convolutional Network (FCN) has the following architecture:

- Encoder (compromised of regular convolutions)
- 1x1 convolution layer
- Decoder (made of reversed convolution layers)
- Skip Connections

Encoder:

The encoder transforms an image input into feature maps. The encoder section is comprised of one or more encoder blocks, each of which includes a separable convolution layer. Each encoder layer allows the model to gain a better understanding of the characteristics in the image. However, each layer is able to gain a better understanding of the image, but more layers increase the computation time.

1x1 Convolution Layer:

The 1x1 convolution layer is a regular convolution, with a kernel and stride of 1. Using a 1x1 convolution layer allows the network to be able to retain spatial information from the encoder and allows the data to be flattened which is useful for classification.

<u>Decoder:</u>

The decoder section of the model can either be composed of transposed convolution layers or bilinear upsampling layers.
The transposed convolution layers reverse the regular convolution layers, multiplying each pixel of the input with the kernel.
Bilinear upsampling uses the weighted average of the four nearest known pixels from the given pixel, estimating the new pixel intensity value. Although bilinear upsampling loses some details, it has much better performance than transposed convolutional layers.
Each decoder layer is able to reconstruct a little bit more spatial resolution from the layer before it. The final decoder layer will output a layer the same size as the original model input image.

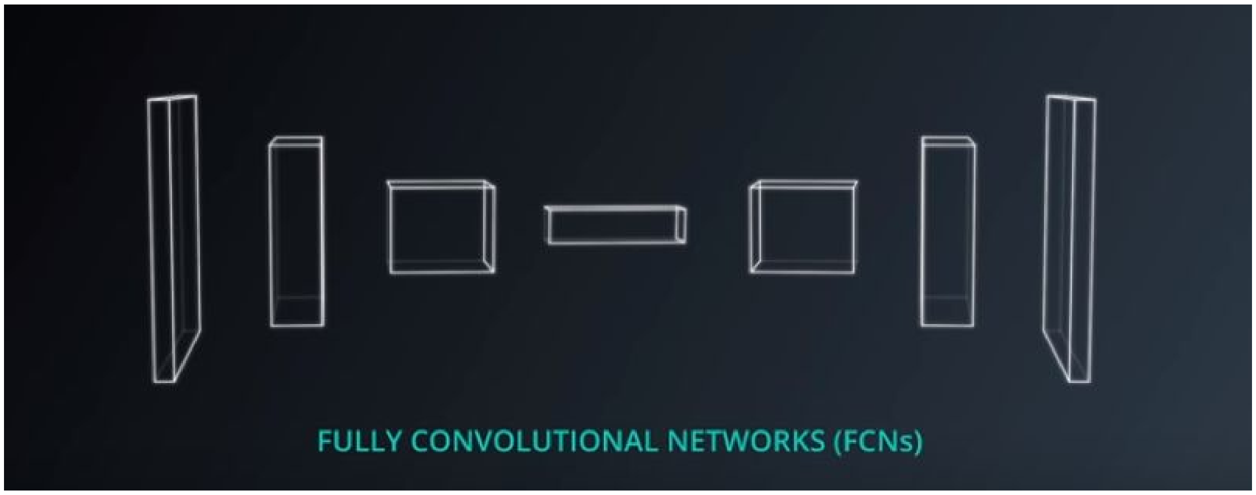
<u>Skip Connections:</u>

Skip connections allow the network to retain information from prior layers that were lost in subsequent convolution layers. Skip layers use the output of one layer as the input to another layer. By using information from multiple image sizes, the model retains more information through the layers and is therefore able to make more precise segmentation decisions.

For this project I've chosen a model with 3 encoder layers, the 1x1 convolution, and then 3 decoder layers. The output shape of each layer is as following:

```
Features Layer (?, 160, 160, 3) - - |
Encoder 1 (?, 80, 80, 32) - - - -|  |
Encoder 2 (?, 40, 40, 64)  - -|  |  |
Encoder 3 (?, 20, 20, 128)    |  |  |  Skip
1x1 Conv (?, 20, 20, 128)     |  |  |  Connections
Decoder 1 (?, 40, 40, 128) - -|  |  |
Decoder 2 (?, 80, 80, 64) - - - -|  |
Decoder 3 (?, 80, 80, 64) - - - - - |
Output Layer (?, 160, 160, 3)
```

Model Used:



The used FCN model consists of three encoder layers, a 1x1 convolution layer, and three decoder block layers.

The first convolution uses a filter size of 32 and a stride of 2, while the second convolution uses a filter size of 64 and a stride of 2 and the third convolution uses a filter size of 128 and a stride of 2. All convolutions used same padding. The padding and the stride of 2 cause each layer to halve the image size, while increasing the depth to match the filter size used.

The 1x1 convolution layer uses a filter size of 128, with the standard kernel and stride size of 1.

The first decoder block layer uses the output from the 1x1 convolution as the small input layer, and the first convolution layer as the large input layer. A filter size of 128 is used for this layer. The second decoder block layer uses the output from the first decoder block as the small input layer, and the original image as the large input layer. This layer uses a filter size of 64. The third decoder block layer uses the output from the second decoder block as the small input layer, and the original image as the large input layer. This layer uses a filter size of 32.

Finally, a convolution layer with softmax activation is applied to the output of the third decoder block.

This is the code for the encoder block:

```
def encoder_block(input_layer, filters, strides):
   # TODO
   output_layer = separable_conv2d_batchnorm(input_layer, filters, strides)
   return output_layer
```

This is the code for the 1x1 convolution layer:

```
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
```

This is the code for the decoder block:

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

<u>**Hyperparameters:**</u>

- **Batch Size**
  As a guiding principle I assume that the lower the batch size is, the noisier the training signal is going to be. On the flip side a higher batch size, it will take longer to compute the gradient for each step. I've then found that the **batch size of 16** seems to be ideal as it results in a stable training signal and offer the best GPU performance.

- **Learning Rate**
  To find the best value of learning rate I’ve tried different values and then compared the loss curves

  Learning rate 0.1:

  - loss: 0.0205
  - val_loss: 0.0268
  - final_score: 0.367229903911

  Learning rate 0.01:

  - loss: 0.0175
  - val_loss: 0.0249
  - final_score: 0.424294565929

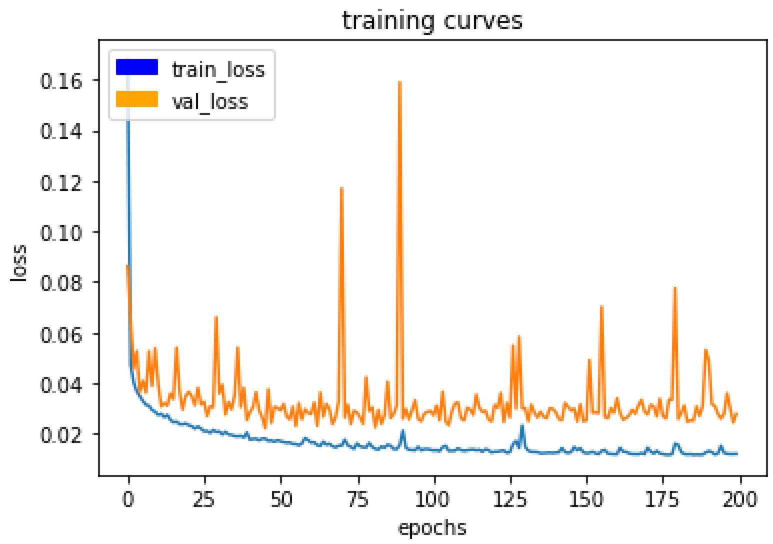
  Learning rate 0.003

  - loss: 0.0178
  - val_loss: 0.0308
  - final_score: 0.382985155838

  Based on the results above I've selected the learning rate of **0.01** because it reached a lower loss and best final score.

- **Number of Epochs**

  I let the model train for 200 epochs (it took over 3 hours on Udacity Workspace) and from the result below we can clearly identify the effects of the overfitting:



  From the curve above we can see that approximately after the **epoch #50** the validation loss stays constant around 0.03 despite constant improvements in the training loss. Since the training beyond that point would only introduce overfitting I've decided to stop there.

So The optimal hyperparameters are:
learning rate = 0.01
batch size = 16
number of epochs = 50
steps per epoch = 200
validation steps = 100
workers = 2

Training:

The model was trained on the amazing Udacity GPU Workspace.

Performance:

While testing the model in the simulator, it performed remarkably well to detect the hero from a reasonably large distance and once it detects and zeros in on the hero, it never lost sight of it. It has a 100% success ratio when it is close to the hero. This is evident from the 539 true positives we obtained from the model evaluation.
The models performance could be improved by using more training data, particularly when the hero is far away. Other than that the model performs really well.

IOU scores

a) Scores for while the quad is following behind the target:
```
number of validation samples intersection over the union evaluated on 542
average intersection over union for background is 0.9964761811370415
average intersection over union for other people is 0.39152376524495486
average intersection over union for the hero is 0.8987985966272466
number true positives: 539, number false positives: 0, number false negatives: 0
```

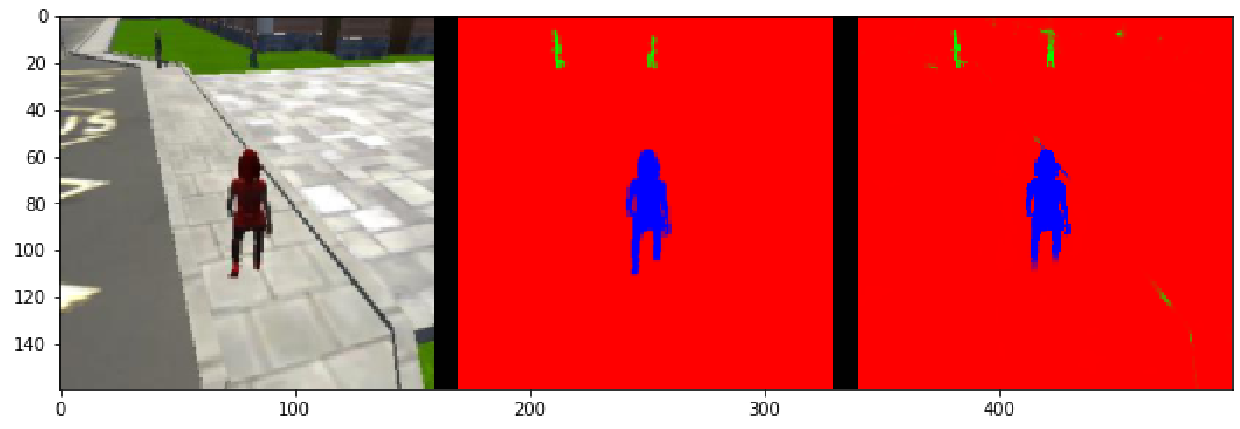
b) Scores for images while the quad is on patrol and the target is not visible:
```
number of validation samples intersection over the union evaluated on 270
average intersection over union for background is 0.9900581698192005
average intersection over union for other people is 0.7971899214466398
average intersection over union for the hero is 0.0
number true positives: 0, number false positives: 23, number false negatives: 0
```

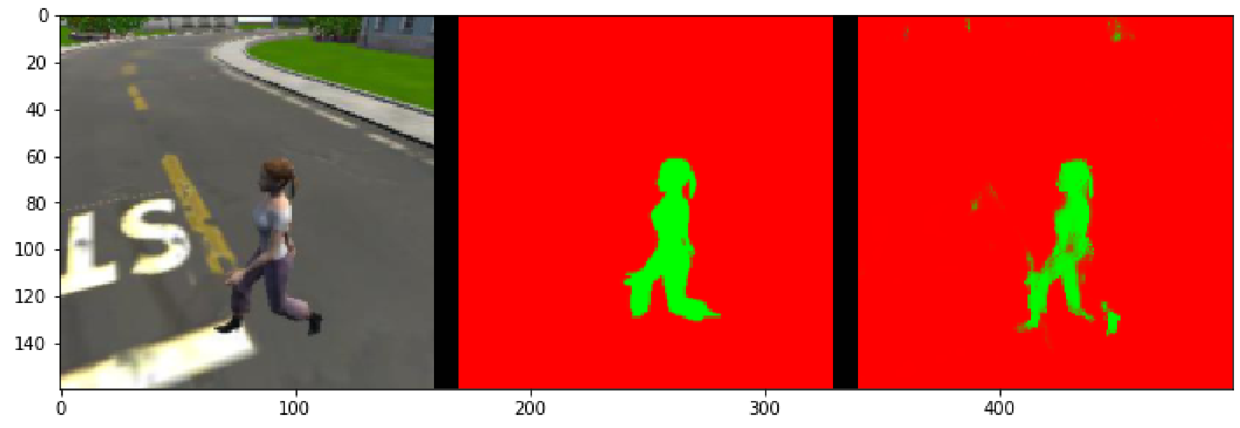
c) Score for images when target from far away:
```
number of validation samples intersection over the union evaluated on 322
average intersection over union for background is 0.9968606185345998
average intersection over union for other people is 0.46960907145941916
average intersection over union for the hero is 0.22014896291105648
number true positives: 117, number false positives: 2, number false negatives: 184
```

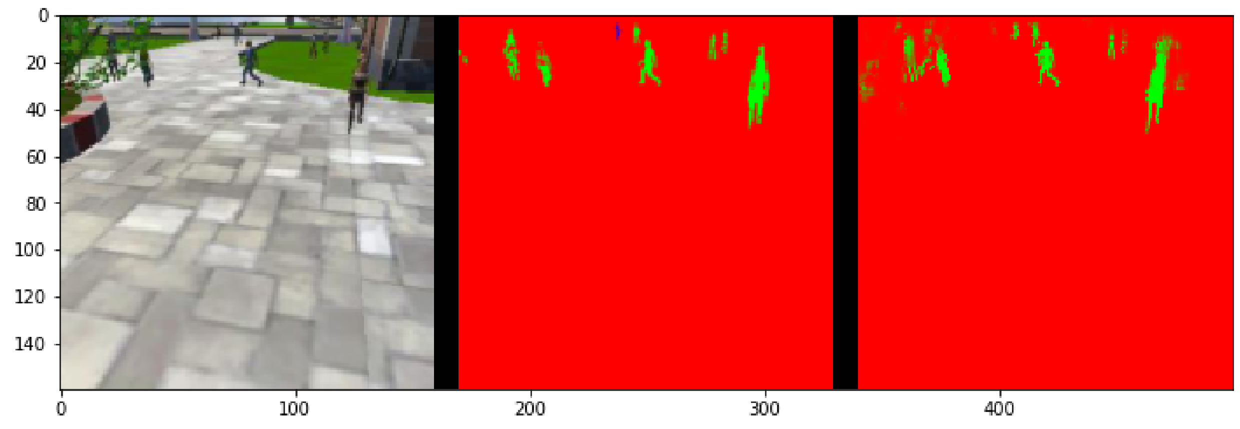
d) Weight = 0.7583815028901734

e) Final IOU = 0.559473779769

f) Final grade score = 0.424294565929







Future Enhancements:

This model was trained on people, however, it could be used to train on any other objects of interest. For example, it could be trained on images of horses or trucks.
I would like to try adding dropout to the model to prevent overfitting as well as pooling layers. Pooling would be interesting as it would provide a better way to reduce the spatial dimensions without losing as much information as convolution strides.