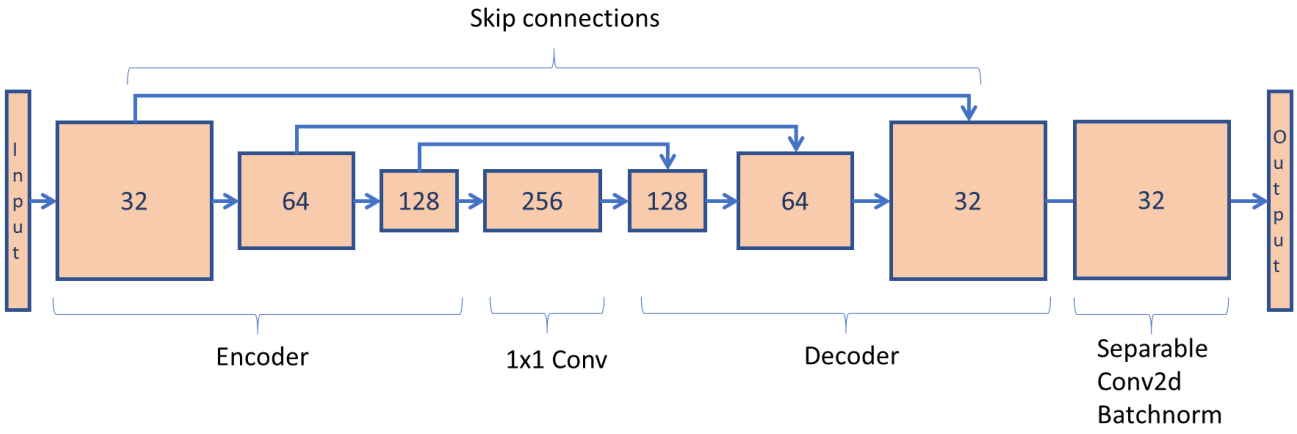
Follow Me Project

The goal is to catch the hero (humanoid with red clothes) in a picture from the drone. Not only to decide that we found the hero in the picture, also to decide where in the picture the hero is located.

1. Network Architecture

A **fully convolutional network** is used for this project containing of an encoder (convolutional layer with batch normalization) followed by a 1x1 Convolution end by the decoder (transposed convolutions). The encoder extract the features of the input images. The 1x1 Convolution layer is used to retain the spatial information of the images instead of a fully connected layer, whereas the decoder layers are used to up sample the previous layer back to the same dimension of the input image. There are also Skip connections, which connect the encoder layers directly to the decoder layers. The advantage out of this is to retain some information which would be otherwise erased, therefore we will have a greater accuracy for our segmentation. The output consists of an convolutional layer with softmax as an activation function to get the three classes.

After several trails and after a suggestions of Udacity support I found the optimal layer as follows:



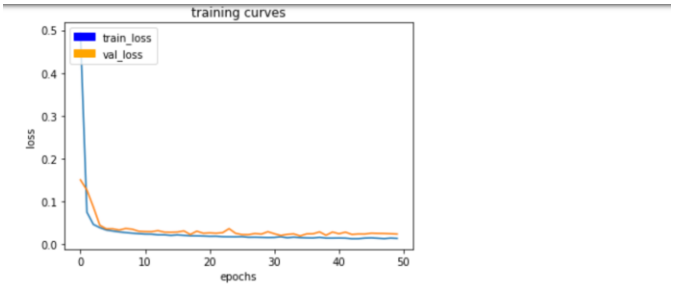
What is not shown in the picture above are the up sampling, concatenating and the separable convolution layer each in the decoder.

```
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_layer1 = encoder_block(inputs, 32, strides=2)
    encoder_layer2 = encoder_block(encoder_layer1, 64, strides=2)
    encoder_layer3 = encoder_block(encoder_layer2, 128, strides=2)

    # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
    Conv1_1_layer = conv2d_batchnorm(encoder_layer3, 256, kernel_size=1, strides=1)

    # TODO: Add the same number of Decoder Blocks as the number of Encoder Blocks
    decoder_layer1 = decoder_block(Conv1_1_layer, encoder_layer2, 128)
    decoder_layer2 = decoder_block(decoder_layer1, encoder_layer1, 64)
    decoder_layer3 = decoder_block(decoder_layer2, inputs, 32)

    decoder_outputLayer = separable_conv2d_batchnorm(decoder_layer3, 32, strides=1)

    # 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')(decoder_outputLayer)
```

2. Training Procedure

I used AWS as a computational machine.

Learning rate: It's in general the step size during gradient descent, the higher the learning rate the faster the training will be whereas a too high learning rate will result in non-convergence. A too small learning rate might result in getting stuck in a local minimum. I played around started with 0.01 which seemed to no find the optimal minimum, there were a lot of jumps in the loss, so I decreased it until 0.001. This was maybe a bit too small (because we do batch normalization we could go with higher learning rate). The loss did not decrease that fast as it did with a higher learning rate. The following picture shows the end result:




```
193/193 [==============================] - 102s - loss: 0.0136 - val_loss: 0.0237
```

Batch size: Determine how many training samples are used in one step of optimization. I chose 64 because I am not limited by my computer.

Number of epochs: I have chosen 22, after I ran with 50 I saw that the model did not get much better.

The steps per epoch and the validation steps are calculated regarding the suggestions.

3. Results

I tried many different thing, listed in the table below:

Network *encoder* (1x1 Conv) "decoder"	**Learning rate**	**Batch size**	**Num epochs**	**Final score**
32/64/128-(256)-"128/64/32"	0.001	32	50	0.397
32/64/128-(512)-"128/64/32"	0.0005	32	50	0.363
64/128/256-(512)-"256/128/64"	0.0005	32	50	0.354
32/64/128/256-(256)-"256/128/64/32"	0.001	50	22	0.337
32/64/128-(256)-"128/64/32"	0.0005	32	50	0.374572
32/64/128-(128)-"128/64/32"	0.001	32	10	0.31
32/64/128-(256)-"128/64/32"	0.001	64	10	0.32

I figured out that the network with 3 layer encoder/decoder works best with filter size of 32/64/128.

Anyway I was not able to go over the 39% So I did collect more data. Furthermore I followed a suggestion of a udacity support member to add a seperable conv2d batchnorm layer at the end of the decoder. Finally I reached a final score of 45% 😊

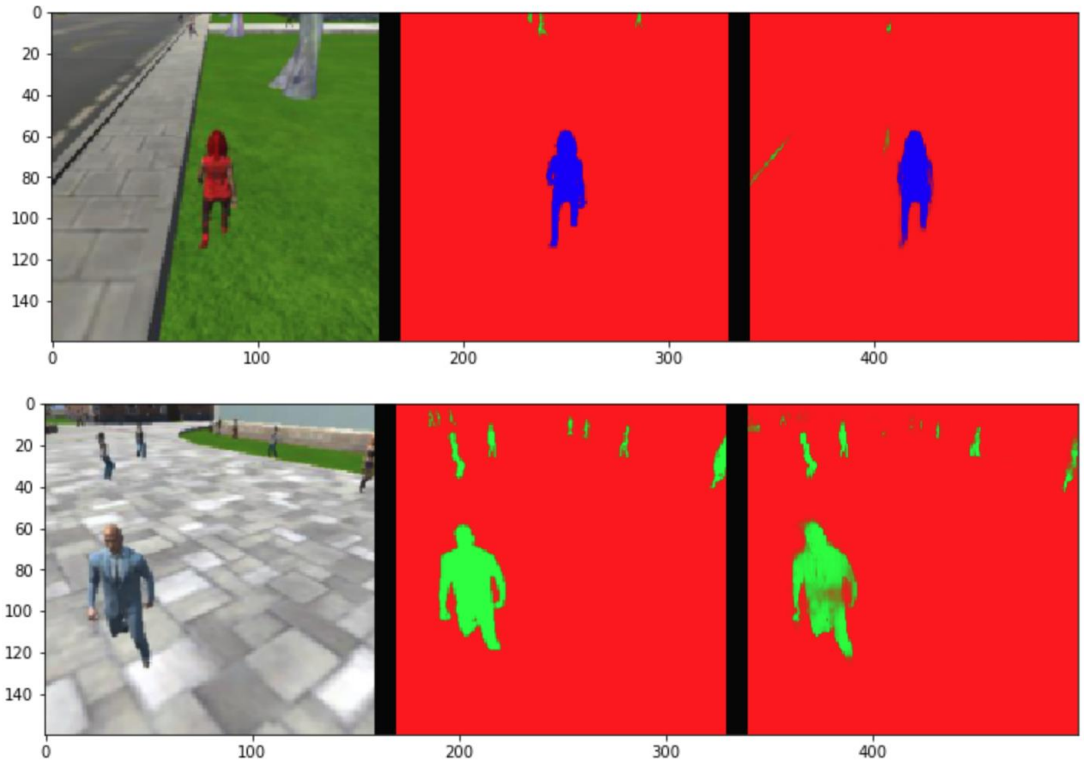
```
learning_rate = 0.001
batch_size = 32
num_epochs = 50
number_train_images = 6207
number_validation_images = 1769
steps_per_epoch = int(number_train_images/batch_size)
validation_steps = int(number_validation_images/batch_size)
workers = 6
print(steps_per_epoch)
print(validation_steps)
```

```
193
55
```

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

```
0.454689802494
```

Finally the drone was able to follow the red clothed person. It would not be possible for the drone to follow other person nor follow some animals or other objects because we simply have not trained them. The data for training was focused on the red clothed person. I we would like to do so like following a car we could use the same architecture but with different input data (focused on a car).



4. Improvement

The filter depth for encoder/decoder could be increased and more pictures of the hero should be catch, in general more data for training would result in a better model.