Project: “Follow me”

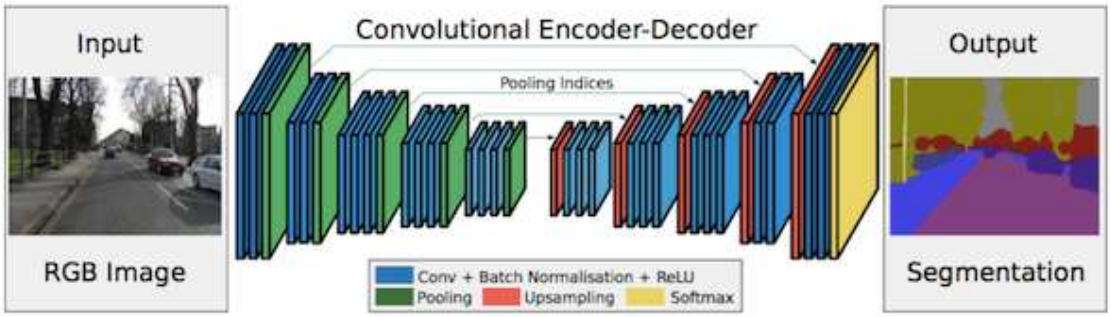
Introduction

This project is about building the deep neural network and training a fully-convolutional neural network to enable a drone to track and follow the ‘hero’ target. In this report I will explain the data collection process, network architecture used, training process and finally discuss the results and possible enhancement.

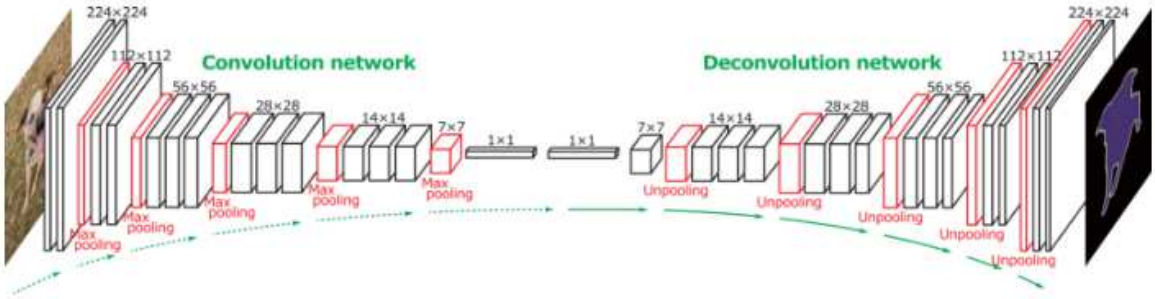
FCN

Fully Convolutional Networks (FCN) don’t have any of the fully-connected layers at the end, which are typically used for classification. This allowed segmentation maps to be generated for image of any size and it is faster compared to the patch classification approach. Apart from fully connected layers, one of the main problems with using CNNs for segmentation is pooling layers. Pooling layers increase the field of view and are able to aggregate the context while discarding the ‘where’ information. However, semantic segmentation requires the exact alignment of class maps and thus, needs the ‘where’ information to be preserved. To tackle this issue the encoder-decoder architecture is applied. Encoder gradually reduces the spatial dimension with pooling layers and decoder gradually recovers the object details and spatial dimension. The 1x1 convolution layer in between extracts the features of the encoders. It adds non-linear classifier to the model, as well as depth and can be very powerful when used with inception. 1X1 convolution helps in reducing the depth dimensionality of the layer. A fully-connected layer of the same size would result in the same number of features. However, with convolutional layers you can feed images of any size into the trained network.

Below there are examples of the FCN architecture.



Source. http://blog.qure.ai/notes/semantic-segmentation-deep-learning-review



Source. Semantic Segmentation using Fully Convolutional Neural Networks

FCNs use convolutional layers to classify each pixel in the image. The final output layer will be the same height and width as the input image, but the number of channels will be equal to the number of classes. Using a softmax probability function, we can find the most likely class for each pixel. The intermediate layers are getting smaller and often deeper, as striding and padding reduce the height and width dimensions of the tensors. The backward convolution layers have the same weights, just like convolutional layers.

Network Architecture

FCN, as was already shown above, consists of encoder blocks, a 1x1 convolution, and decoder blocks.

The steps are:

1. Replace the fully connected layers by 1x1 convolution layer
2. Upsampling through the transpose convolutional layers
3. Skip connections, which allows to use information from multiple resolution scales

Encoder blocks

Encoder takes an input image and generates a high-dimensional feature vector and aggregates features at multiple levels. This encoder part have pooling which down-samples the image. The model reduces the risk of overfitting, however loses the information and reduces the spatial dimension. The encoder section consists of one or more encoder blocks, where each includes a separable convolution layer. Each encoder layer allows the model to gain a better understanding of the shapes in the image. For example, the first layer is able to discern very basic characteristics in the image, such as lines, hues and brightness. The next layer is able to identify shapes, such as squares, circles and curves. However, the downside of this is that it loses information and reduces the spatial dimension.

1x1 Convolution Layer

By using a 1x1 convolution layer, the neural network is able to take retail the spatial information from the encoder. When using a fully connected layer, the data is flattened, retaining only 2 dimensions of information. Flattening the is useful for classification, however, the model needs to be able to classify each pixel in the image. 1x1 convolution layers allow the network to retain this location information. The extra depth to the model can be added, but this convolution layer with a kernel and stride of 1.

Decoder block

Decoder takes a high dimensional feature vector and generates a semantic segmentation mask and decode features aggregated by encoder at multiple levels. The decoder part gradually recovers the spatial dimension. 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 finer details

when compared to transposed convolutions, it has much better performance, which is important for training large models quickly. Each decoder layer is able to reconstruct the spatial resolution from the previous layer. The final decoder layer will output a layer the same size as the original model input image, which will be used for the quad.

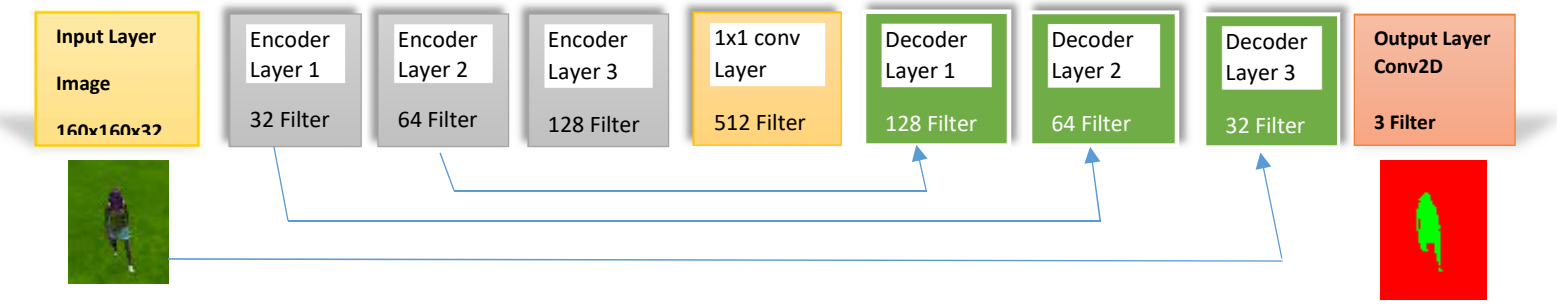
Skip Connections

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 is able to make more precise segmentation decisions.

Building FCN requires experimentation with different numbers of layers, filter sizes and hyper parameters.

In my model I have used three layers of encoder and decoder. The 2 layers were not sufficient to obtain the target score (at least the the chosen parameters).

```
encoder_layer1 = encoder_block(inputs, 32, 2)
encoder_layer2 = encoder_block(encoder_layer1, 64, 2)
encoder_layer3 = encoder_block(encoder_layer2, 128, 2)
conv_layer = conv2d_batchnorm(encoder_layer3, 512 ,kernel_size=1,strides=1)
decoder_layer1 = decoder_block(conv_layer, encoder_layer2, 128)
decoder_layer2 = decoder_block(decoder_layer1, encoder_layer1, 64)
decoder_layer3 = decoder_block(decoder_layer2, inputs, 32)
```



In the model the network takes as input a 160x160x32 input image and passes that image through 3 encoders (blocks). Each encoder applies a different number of 3x3 kernels with a stride of 2 followed by batch normalization.

To connect the encoder layers to the decoder, instead of a fully connected layer, we use a 1x1 convolutional layer with 512 filters to retain the spatial information. Upsampling in the decoder part used bilinear interpolation for the better results. The output of the 1x1 convolution is then fed into the three decoders. Each decoder up-samples the input by a factor of 2, concatenates the result with an input from the encoder stage using a skip connection and applies two convolution and batch normalization layers using a different number of 3x3 kernels with a stride of 1. The output of the last decoder has dimensions 160x160x32. This tensor is passed into a final convolutional layer that is responsible for producing an image segmentation of dimension 160x160x3 (3 channels for each segmentation class).

Choosing Hyper parameters

After we have built the network we pick the hyper parameters:

Learning Rate: It is important to find the balance between the learning rate and the number of epochs. By increasing the accuracy the learning rate the test accuracy might drop. Lowering the learning rate would require more epochs, but could ultimately achieve better accuracy.

After a trial, where I experimented with 0.05 accuracy, I have finally chosen a learning rate of 0.001. With this rate the network learning could progress at a reasonable rate without the risk of failing to converge and showing good results.

Batch size: Batch size is mainly depended to the memory in GPU/RAM. It has been observed in practice that when using a larger batch there is a significant degradation in the quality of the model, as measured by its ability to generalize.

I chose a batch size of 32 so as to avoid having very noisy gradient estimates as well as to well utilize the GPU (and avoid excessive copying of data from host CPU to GPU).

Number of Epochs: 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. Each epoch attempts to move to a lower cost, leading to better accuracy.

Lowering the learning rate would require more epochs, but could ultimately achieve better accuracy. I initially trained the network for 50-40 epochs and found that my score was quite low. Training the network with 100 epochs led to the better score, 0.42.

Steps per Epoch: This is the number of steps (batches of samples) before declaring that the epoch finished. This should be the number of training images over batch size because it should theoretically train the entire data on very epoch.

I had initially 4131 training images and I wanted each epoch to pass all the data through the network. With a batch size of 32, the number of steps per epoch was 130.

Validation Steps: This should be number of validation images over batch size because it should test all the data on every epoch. I had initially approximately 1000 validation images, and the validation score to encompass all of them. With a batch size of 32, the number of validation steps calculated as 30.

Workers: This is the number of parallel processes during training. It states that this can affect the training speed and is dependent on the hardware. I tried a 2, 4, 10.

The Optimal Hyperparameters for my model was found as:

Learning rate = 0.001
Batch size = 32
Number of epochs = 100
Steps per epoch = 130
Validation steps = 30
Workers = 2

Lowering the learning rate made and increasing the number of epochs with the same batsch size made it possible to achieve 40% of the score overall IoU.

Training

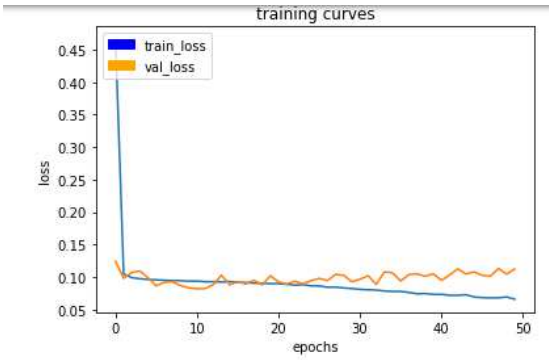
I trained the model on AWS using a p2.xlarge instance.

The training images are from a default data set given by Udacity which are taken from images of cameras captured by a drone in the simulator. To measure how well the model is performing, the IOU metric is used which takes the intersection of the prediction pixels and ground truth pixels and divides it by the union of them. Only the images with the people (no other objects will be recognised) are evaluated with this metric. Because we have used the model with classified images of humans from the simulator, the model is taught to distinguish between the human target in red clothes and the other people in the simulator.

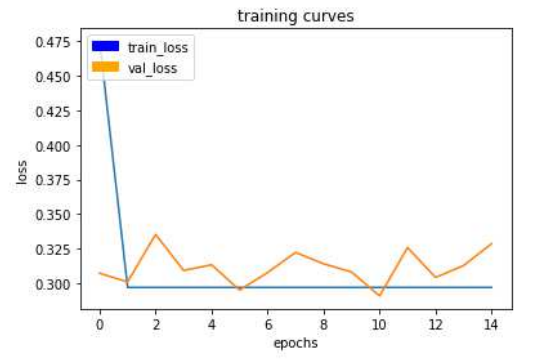
Here are some test images of the model output. The left frame is the raw image, the middle frame is a goal, and the right frame is the output from the model. The objective was to have the right frame as close as possible to the middle frame. If the target person (hero) is identified and has a blue colour. The non-target people are yellow.

Trial N^{0}n: (no success)

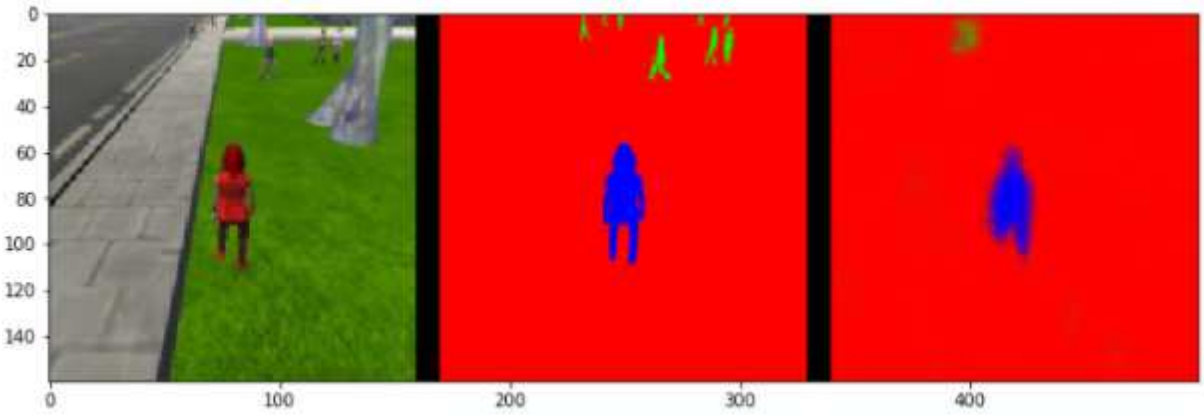
When following the target at close range, the model had a score of only 0.05 and final IoU 0.55. The frames below show the model's output:



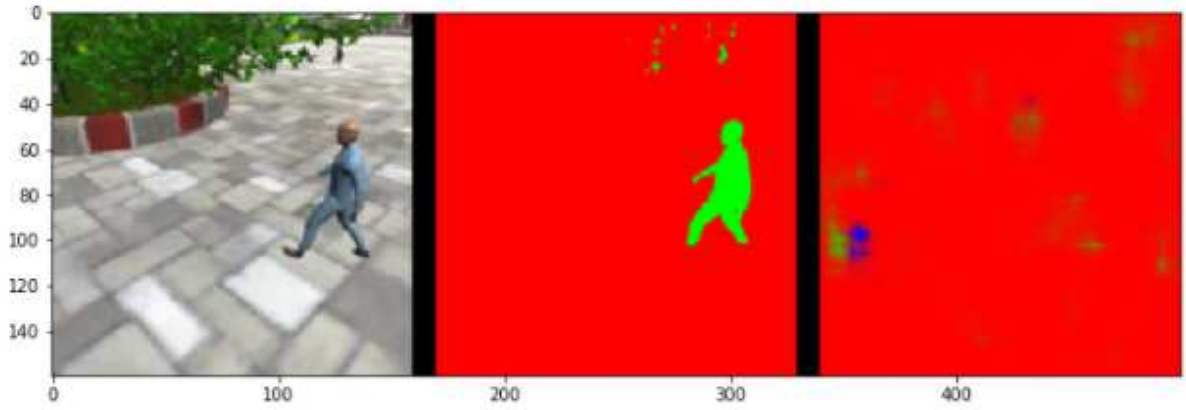
number of epoch, 50



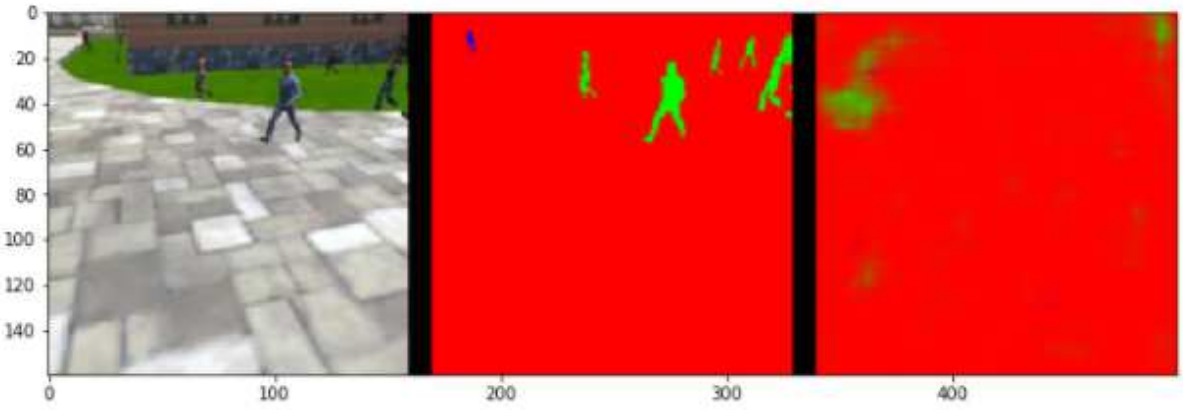
learning rate, 0.05



images while following the target



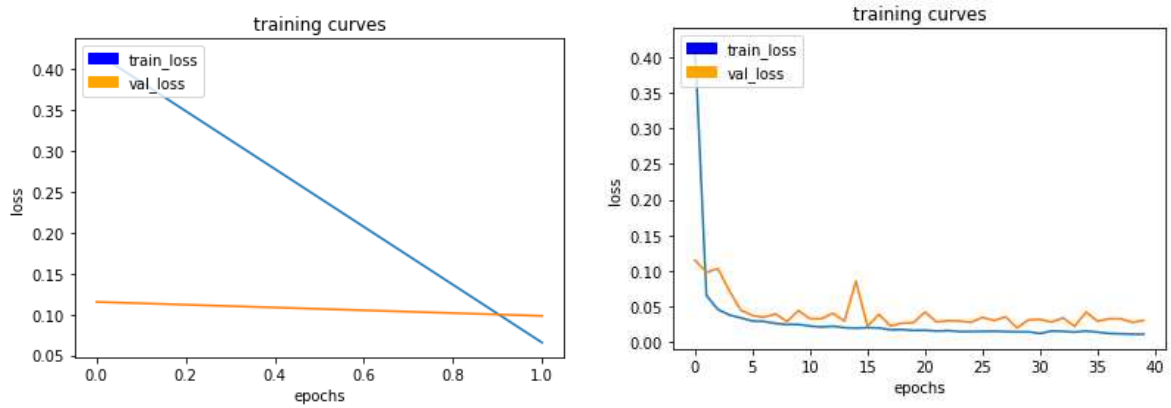
images while at patrol without target



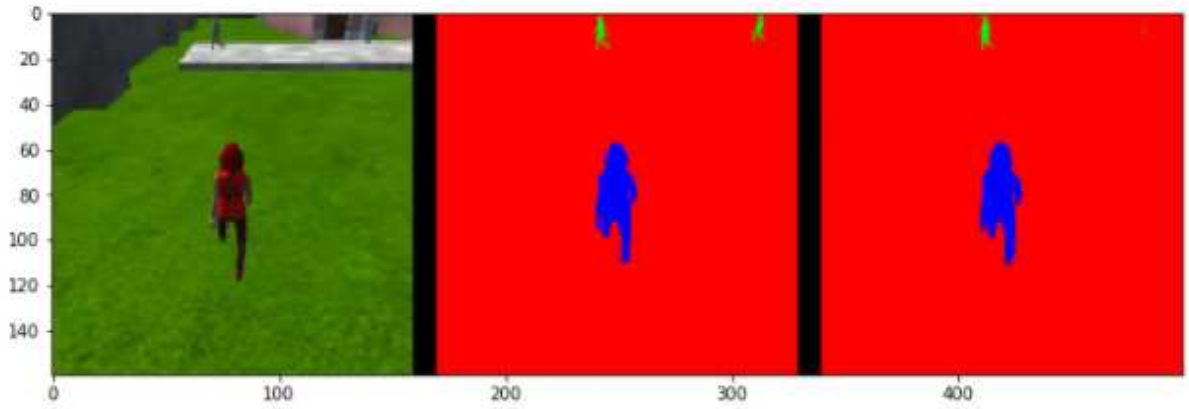
images while at patrol with target

Trial 1+n (improved, but still no success)

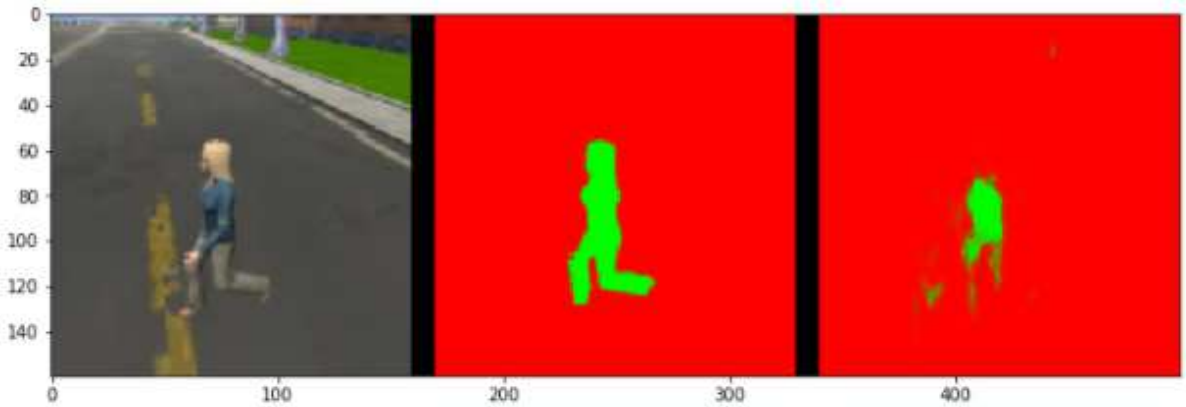
When following the target at close range, the model had a score of only 0.385 and final IoU 0.527



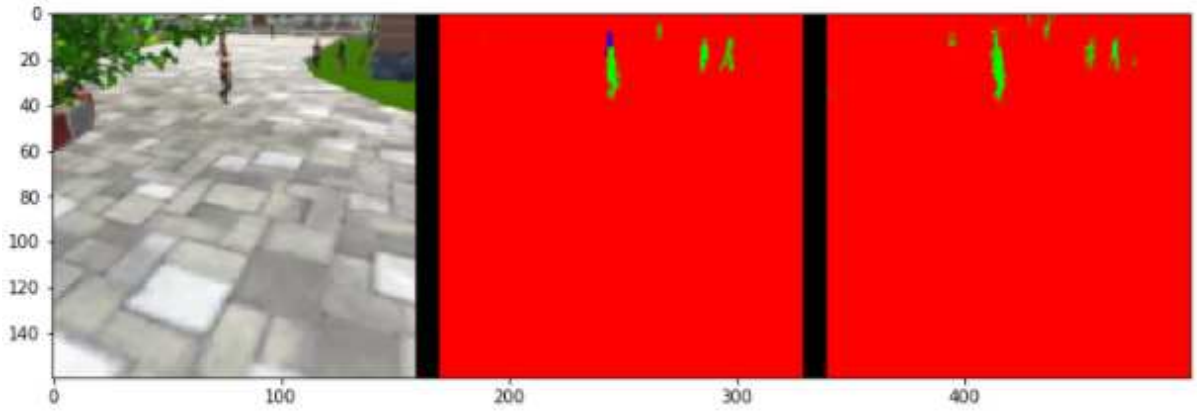
Learning rate 0.002, number of epoch 40



images while following the target



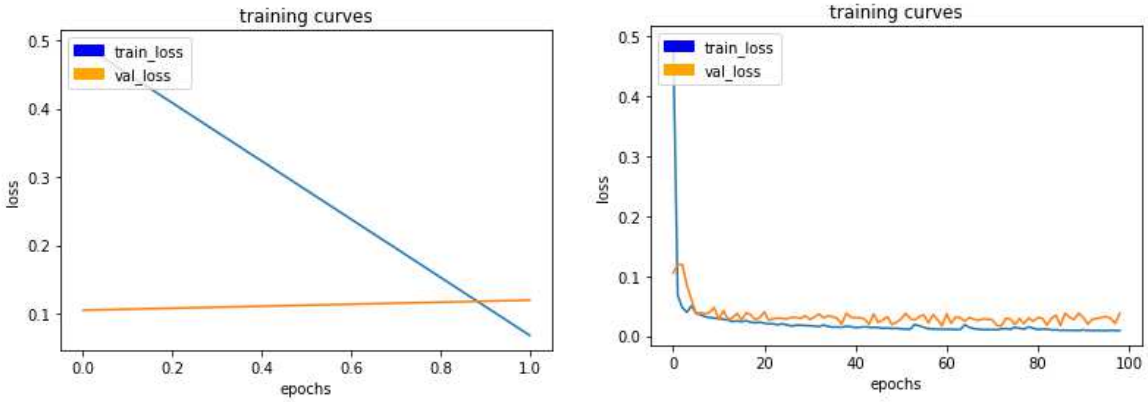
images while at patrol without target

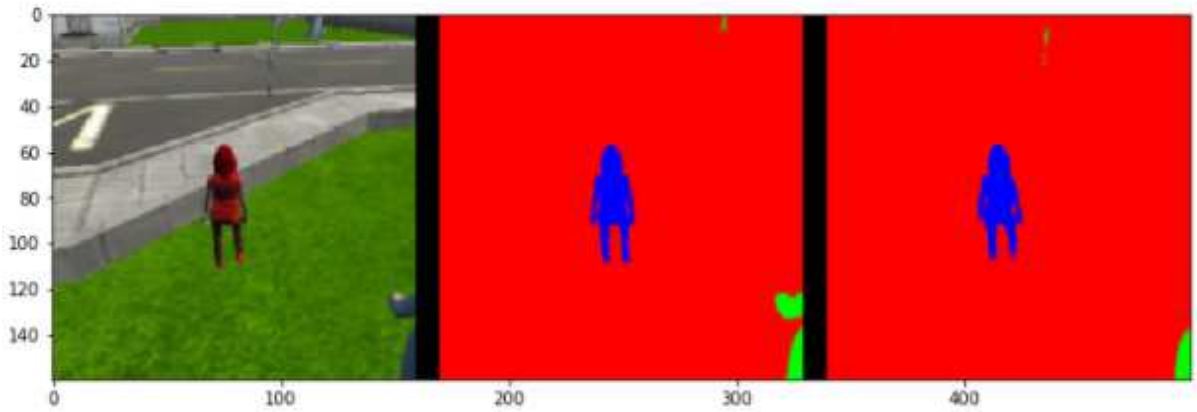


images while at patrol with target

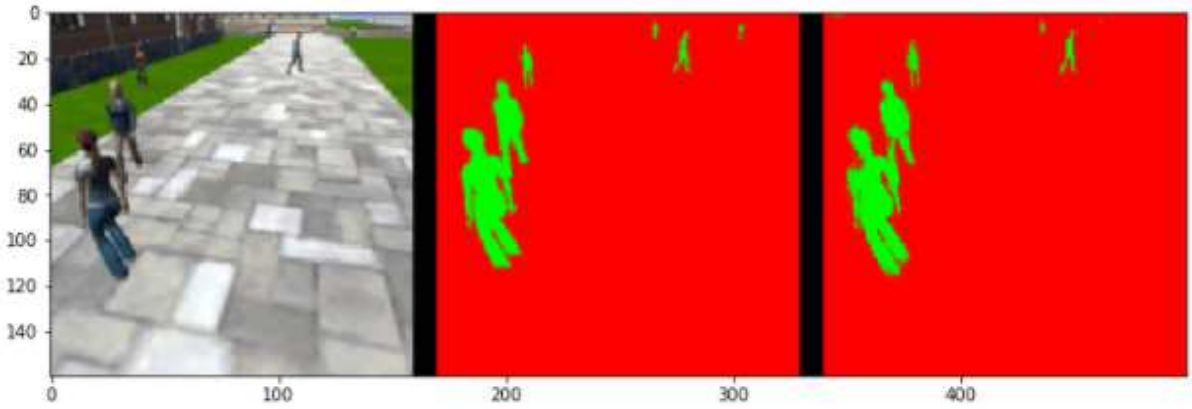
Final trial (success)

The final score of my model, after several trials, was **0.421495187757**, while the final IoU was **0.5842322351**. The figures are shown below.

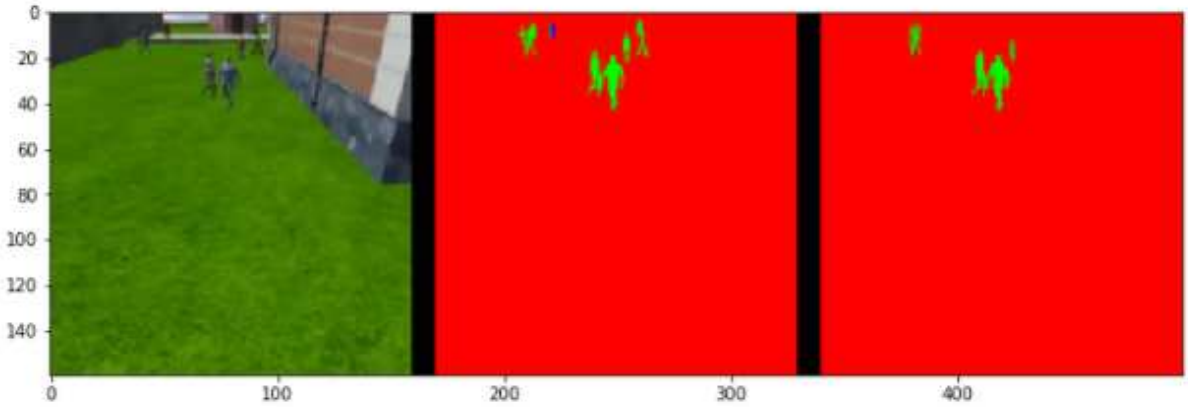




images while following the target



images while at patrol without target



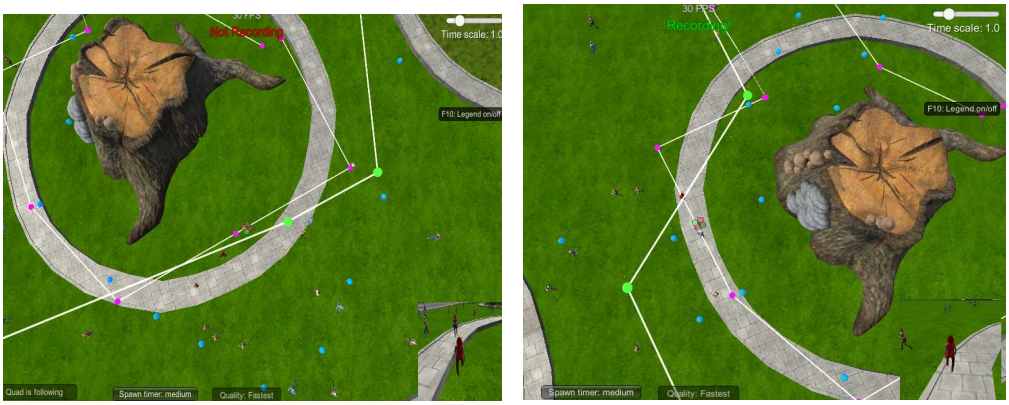
images while at patrol with target

Note:

Given the current data, the model would not work for a different object like car or animal. The system needs this kind of data to be trained and tested to follow. The neural network is highly dependent on the data it is being trained on. Once that is done the network can output a channel for each target class (e.g. one channel for the hero, another for roads, a third for tree etc).

Simulation

Using simulator I have recorded some more data to train the network.



In “Follow me!” mode in RoboND Quad Sim simulator mode the quad was able to find and follow the “hero”.

Discussion and Future Enhancements

The simulation works well and achieves the intended result, however the final IOU scores seem a little low. It could be possible to improve these scores with an increase of training data, especially related to other people. As it is seen from the long range images, the model has difficultly at long distances. Deep networks need large amount of training data to achieve good performance. So additional training images date set would improve the results and decrease the error.

Another approach is the data augmentation via a number of random transformations, so that the model would never see twice the exact same picture. This helps prevent overfitting and helps the model generalize better. (Image Augmentation for Deep Learning With Keras)

As an enhancement, it is also would be possible to consider the encoding with different kernel sizes.

However, the model was good enough that the quad had no trouble finding and following the target person in the simulated environment.