Project Introduction

In this project, we will build our own segmentation network, train it, validate it, and deploy it. In the end, we will have our own drone tracking and following a single hero target using our own network pipeline!

Segmentation task is different from classification task because it requires predicting a class for each pixel of the input image, instead of only 1 class for the whole input. Classification needs to understand *what* is in the input (namely, the context). However, in order to predict what is in the input for each pixel, segmentation needs to recover not only *what* is in the input, but also *where*. We use fully convolutional neural layer architecture to achieve the same.

Project Steps:

1 Setting up your local environment: This is how you will test your initial network design to make sure there are no errors. Otherwise, you can use up your Workspace GPU hours or rack up charges on Amazon Web Services (AWS) with simple debugging instead of training. You will also use your local environment to evaluate your Workspace/AWS trained model with the Follow Me simulator.
2. Brief overview on how the simulator works and its basic controls.
3. Collecting data from the simulator to train your network.
4. Building your neural network.
5. Setting up your Classroom Workspace or, if you prefer, setting up your AWS Amazon Machine Images (AMI).
6. Training your network and extracting your final model and weights from Udacity GPU Workspace or AWS Instance.
7. Downloading your model from cloud and testing your model with the Follow Me simulator.
8. Getting your project ready for the final submission!

Here I am going to illustrate some of the crucial stages while following the above project guidelines:

Data collection:

Data can be collected by using the RonoNd Quad sim tool. I have followed following steps for data collection:

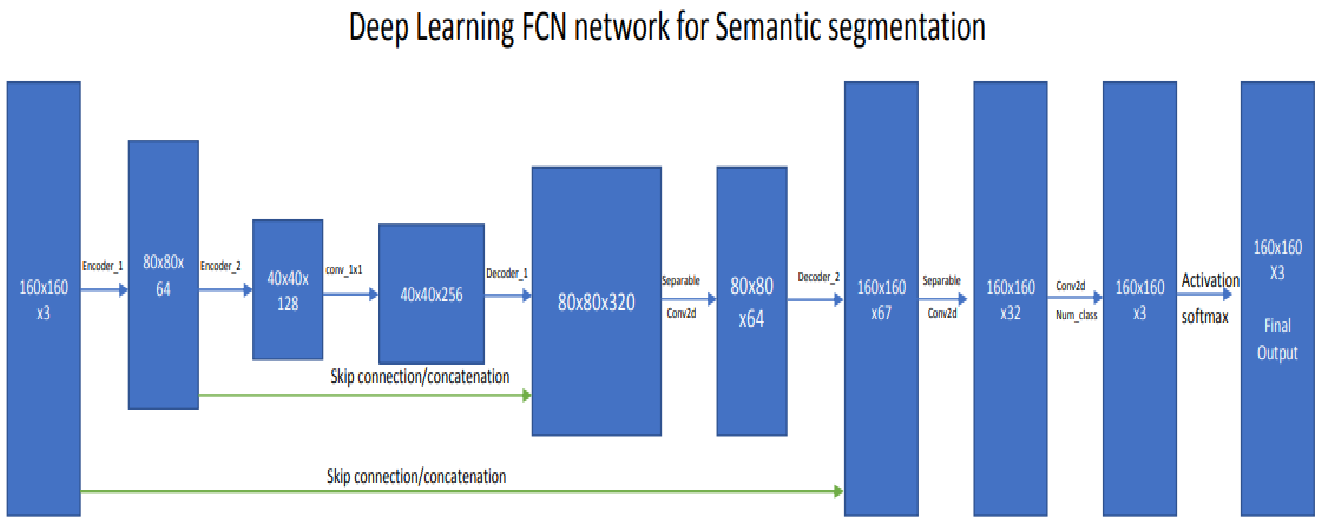
1. Open the windows based Quad sim tool.
2. Click on “DL training” while making sure the “spawn crowd” box is ticked.
3. Use ‘H’ button for local control mode.
4. To set petrol points, use ‘P’ button. Use ‘H’ to enter into petrol mode.
5. Use ‘O’ button to set Hero path points.
6. Use ‘I’ button to set spawn points.
7. Set many petrol, Hero path and Spawn points and press ‘M’ to start people spawning.
8. Press ‘R’ to start recording. Use number keys ‘1’ to ‘9’ to set speed of data recording.
9. Select folder path “\data\raw_sim_data\train\run1” for collecting training data sets.
10. Similarly, select folder path “\data\raw_sim_data\validation\run1” for collecting validation data sets.
11. Try various runs of data collections by following above steps and using different runx folders.

12. Open the command prompt and activate RoboND environment.
13. cd to \code folder and type python ***preprocess_ims.py*** to preprocess all the recorded data sets into a format which can be fed to FCN code.
14. The preprocessed training data and validation data will be stored in \data\processed_sim_data\train and \data\processed_sim_data\validation folders respectively. You can see that subfolders "image" and "mask" are created under each of this folder. The images in" mask" folder represents the labelled data sets for training and validation of the network.
15. Move these folders (preprocessed sets of images) to \data\train and \data\validation folder respectively.

This completes the data recording steps.

First, I used the data sets provided in the default directory and tried to learn the deep learning network over it.

Building the Neural Network:



As explained in the classroom lectures, we will use an FCN (fully convolutional network) for semantic segmentation. FCN has a wide use in scene understanding and object classification inside an image. The basic difference in a FCN vs CNN is that we do not use a fully connected layer at the end. In a normal CNN architecture, we flatten the layer at the end of few convolutional layers (convolution + Max pooling) and then apply a linear classifier with SoftMax activation. While we do this in CNN, we loose spatial integrity of the data during the flattening step and hence semantic segmentation is not possible with CNN architecture. But, We can achieve object classification for entire image with CNN architecture.

In order to classify object within the image which is also referred as sematic segmentation, we need to preserve the spatial integrity of the data. In order to do so, we do not apply a fully connected neural network at the end, rather we apply 1x1 convolutions. By doing 1x1 convolutions, we can preserve the spatial information embedded in each pixel while changing the depth of the image by applying suitable 1x1 convolution filer size. 1x1 convolution also allows us to feed any size of the input image to the network architecture and be able to compute sematic segmentation.

The basic architecture of a fully convolution layer network consists of following three steps:

Encoder

1x1 convolution

Decoder

Skip connections

Encoder block:

The encoder block includes various convolutional layers with batch normalization.

Depth wise separable convolutions:

It is better to use depth wise separable convolutional layers rather than simply using normal convolutional layers. A depth wise separable convolution splits a kernel into 2 separate kernels that do two convolutions: the depth wise convolution and the pointwise convolution. The main advantage of using depth wise separable convolutional layer is that it enhances the compute efficiency by reducing the number of parameters. The depth wise convolution uses different kernel for each channel of the image and then sums it up. The point wise convolution uses a 1x1 kernel on the depth wise convoluted image. We can obtain various depth of the output image by using various 1x1 convolution filter. Because it reduces the number of parameters in a convolution, if your network is already small, you might end up with too few parameters and your network might fail to properly learn during training. If used properly, however, it manages to enhance efficiency without significantly reducing effectiveness, which makes it a quite popular choice.

Batch Normalization:

We know that normalizing the inputs to a network helps it learn. But a network is just a series of layers, where the output of one layer becomes the input to the next. That means we can think of any layer in a neural network as the first layer of a smaller subsequent network.

To increase the stability of a neural network, batch normalization normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. This helps in quick convergence of the network. We can also use a higher learning rate hyperparameter if we use batch normalization.

Decoder block:

The decoder block consists of transposed convolutions, layer concatenation through skip connections and separable convolutions with batch normalization.

We can use transposes convolution to conduct up-sampling. Moreover, the weights in the transposed convolution are learnable. So, we do not need a predefined interpolation method. We up-sample the input by adding zeros between the values in the input matrix in a way that the direct convolution produces the same effect as the transposed convolution. We are using bilinear up sampling in this project. It works similar to pooling concept. We interpolate values between the input entries along width and height and form a up sampled image.

Skip connection:

To fully recover the fine-grained spatial information lost in the pooling or encoder layers, we often use skip connections. A skip connection is a connection that bypasses at least one layer. Here, it is often used to transfer local information by concatenating or summing feature maps from the down sampling path with feature maps from the up sampling path. Merging features from various resolution levels helps combining context information with spatial information.

Workspace code:

Based on the above FCN architecture, we train the model asshown below :

```
def fcn_model(inputs, num_classes)

   encoder_1 =  encoder_block(inputs, 64,strides =2)

   encoder_2 =  encoder_block(encoder_1, 128,strides =2)

   conv_1x1 = conv2d_batchnorm(encoder_2, filters = 256, kernel_size=1, strides=1)

   decoder_1 = decoder_block(conv_1x1, encoder_1, 64)

   decoder_2 = decoder_block(decoder_1, inputs, 32)

   return layers.Conv2D(num_classes, 3, activation='softmax', padding='same')(decoder_2)
```

As described in the layer architecture diagram, we apply two sets of encoder layers followed by 1x1 convolution layer with filter size of 256. The output of the 1x1 convolution layers fed to two layers of decoder layer consecutively. In each decoder layer, we perform skip connection to transfer information lost during encoder stages. Finally, the decoder layer output goes through a convolution layer with

SoftMax activation. Each pixel wise prediction is performed in subsequent block of code to identify the hero in the scene.

Hyper parameters:

Each of the following hyper-parameters are found by performing several iterations of the training cycles. Number of Epochs reflects the number of forward and backward propagation path that the network performs for optimizing the parameters. Increasing this parameter increases the compute time but increases the accuracy of the network. Optimum size of batch size was determined to enhance. computation complexity. Finally, I came out these sets of parameters.

learning_rate = 0.001

batch_size = 32

num_epochs = 100

steps_per_epoch = 200

validation_steps = 50

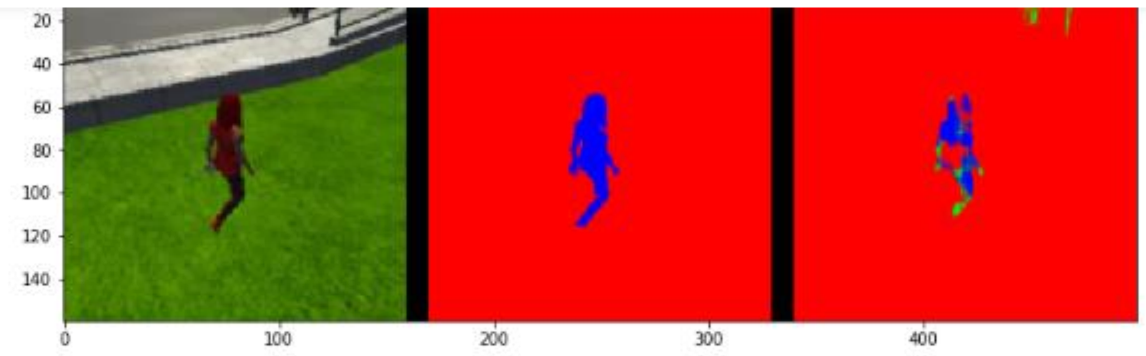
workers = 4

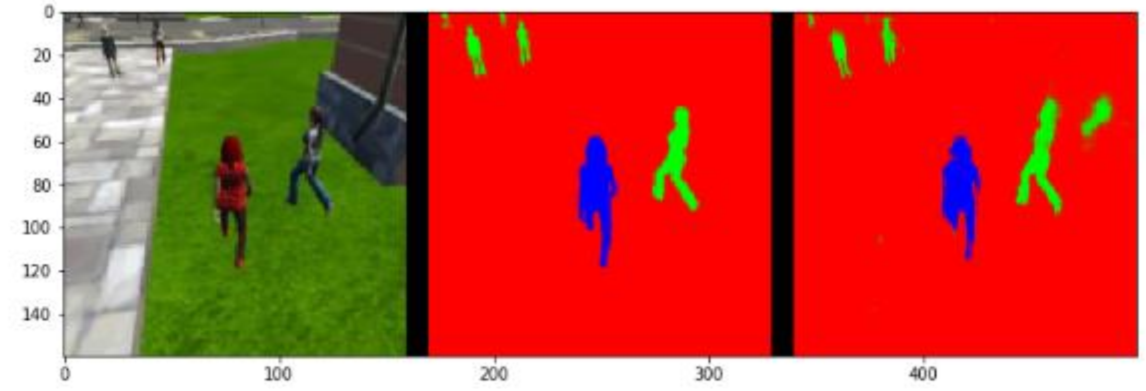
Workspace:

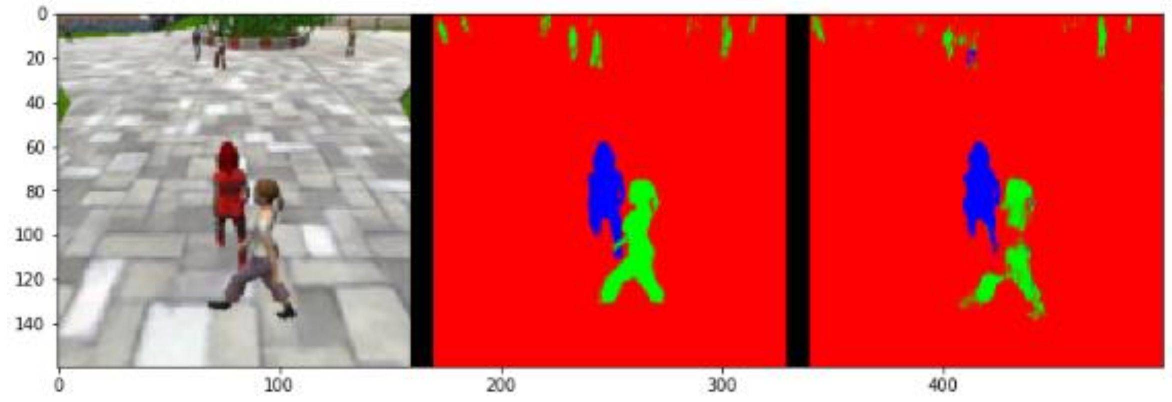
I used the Udacity classroom workspace to check the prediction with given datasets. I also set up AWS EC2 instance and trained the model.

Images with evaluation data:

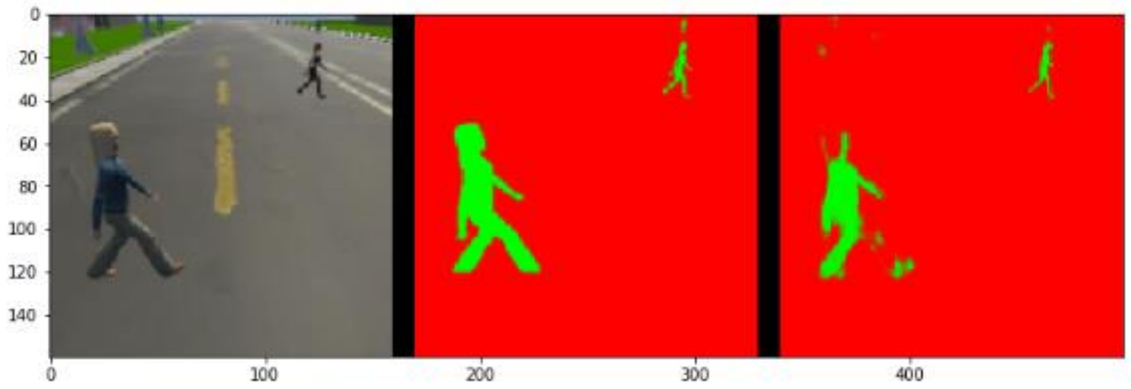
Images while following the target:

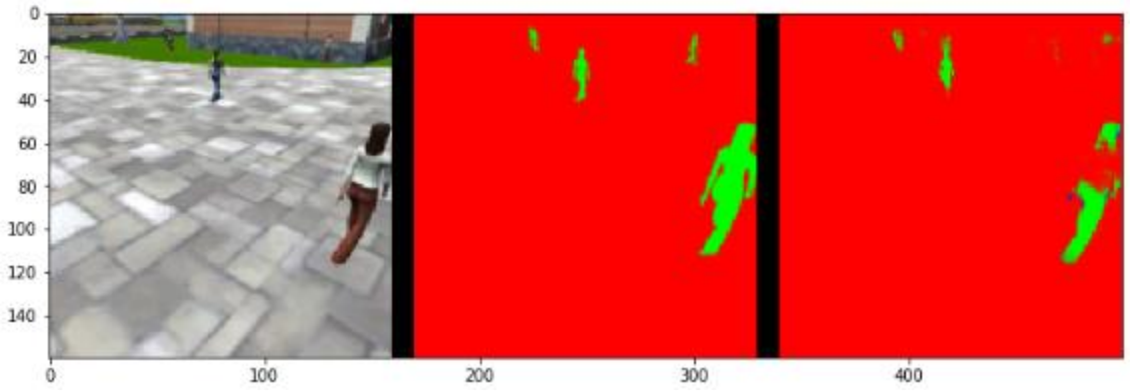

20
40
60
80
100
120
140
0
100
200
300
400

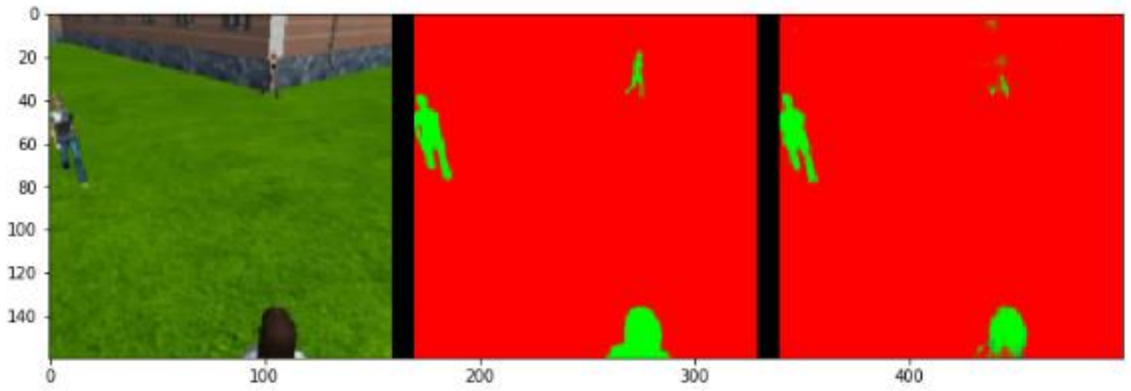

0
20
40
60
80
100
120
140
0
100
200
300
400


0
20
40
60
80
100
120
140
0
100
200
300
400

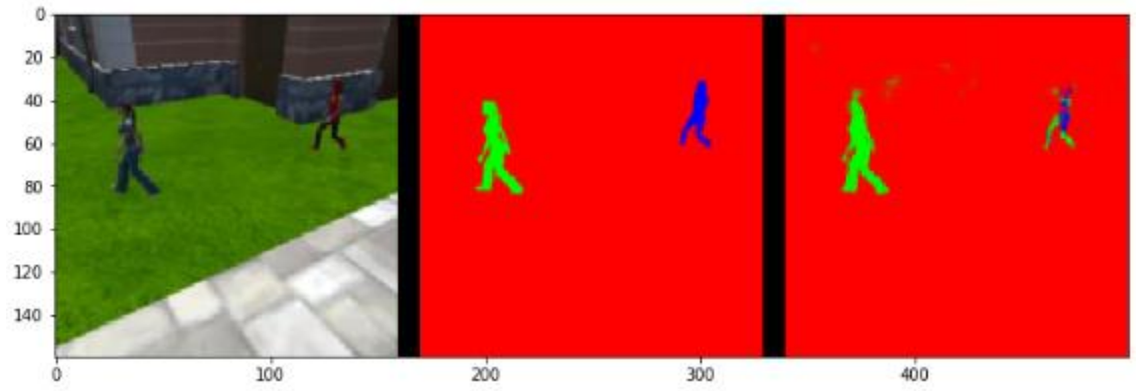
Images while petrol without target:

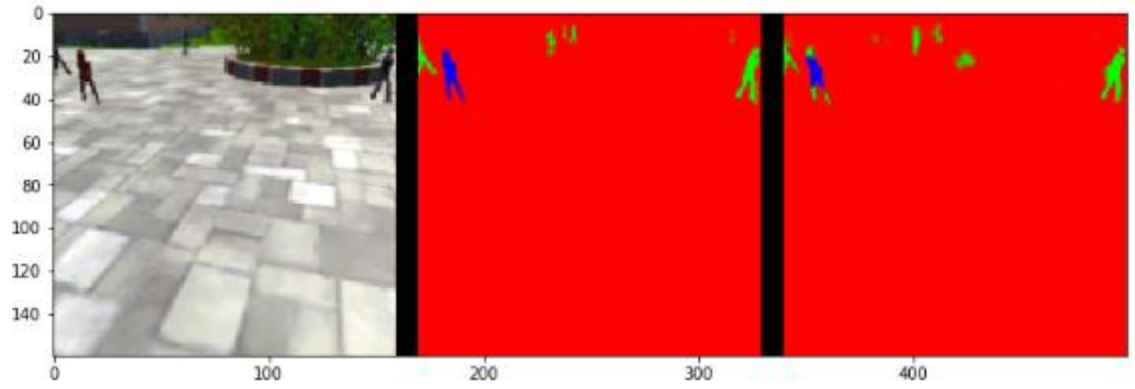


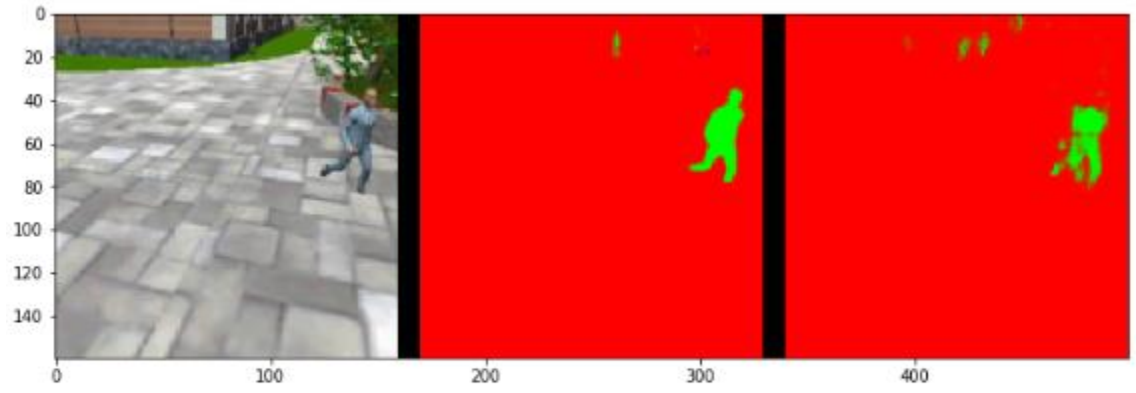




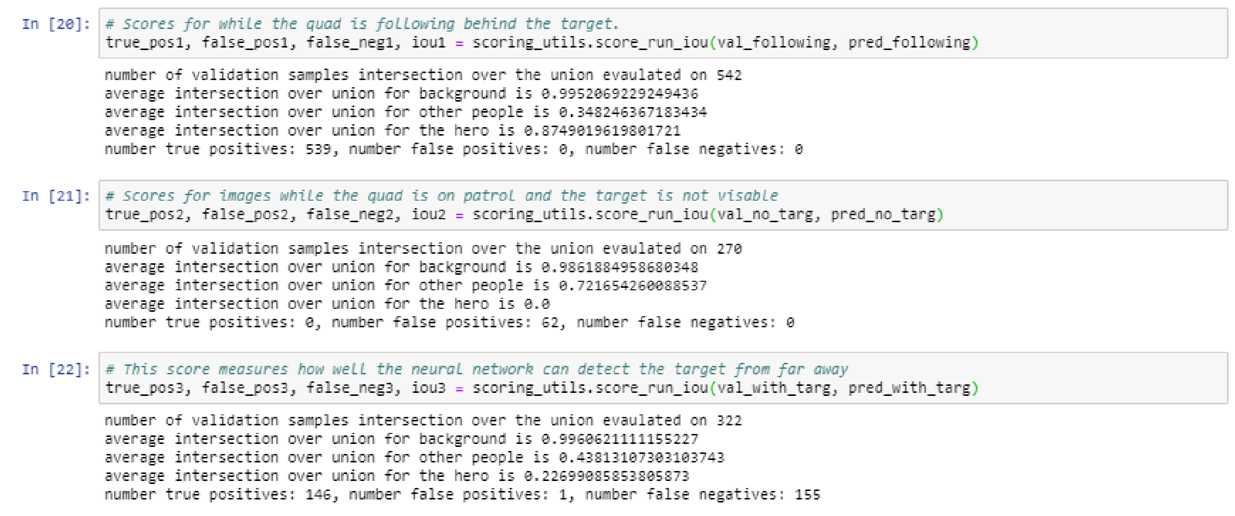
Images while at petrol with target:



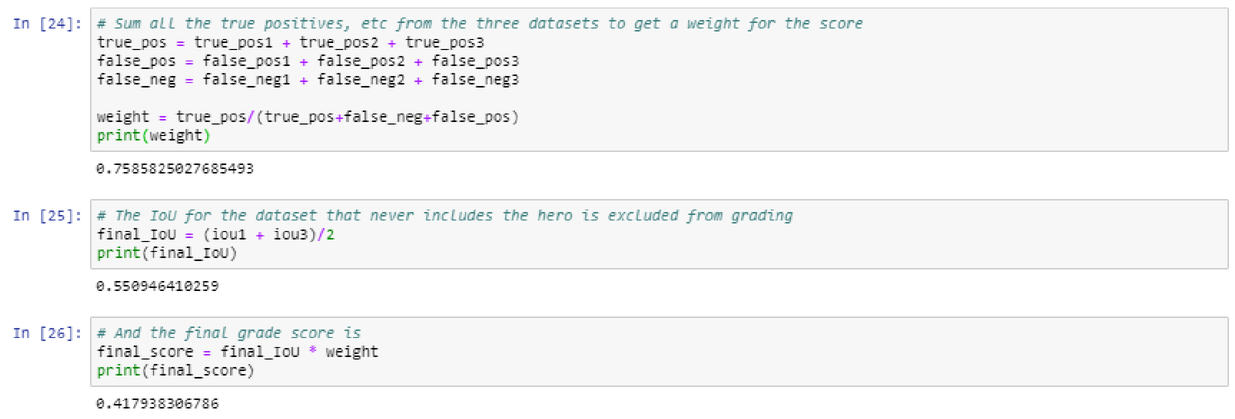




Evaluation Results:



```
In [20]: # Scores for while the quad is following behind the target.
         true_pos1, false_pos1, false_neg1, iou1 = scoring_utils.score_run_iou(val_following, pred_following)

         number of validation samples intersection over the union evaulated on 542
         average intersection over union for background is 0.9952069229249436
         average intersection over union for other people is 0.348246367183434
         average intersection over union for the hero is 0.8749019619801721
         number true positives: 539, number false positives: 0, number false negatives: 0

In [21]: # Scores for images while the quad is on patrol and the target is not visable
         true_pos2, false_pos2, false_neg2, iou2 = scoring_utils.score_run_iou(val_no_targ, pred_no_targ)

         number of validation samples intersection over the union evaulated on 270
         average intersection over union for background is 0.9861884958680348
         average intersection over union for other people is 0.721654260088537
         average intersection over union for the hero is 0.0
         number true positives: 0, number false positives: 62, number false negatives: 0

In [22]: # This score measures how well the neural network can detect the target from far away
         true_pos3, false_pos3, false_neg3, iou3 = scoring_utils.score_run_iou(val_with_targ, pred_with_targ)

         number of validation samples intersection over the union evaulated on 322
         average intersection over union for background is 0.9960621111155227
         average intersection over union for other people is 0.43813107303103743
         average intersection over union for the hero is 0.22699085853805873
         number true positives: 146, number false positives: 1, number false negatives: 155
```



```
In [24]: # Sum all the true positives, etc from the three datasets to get a weight for the score
         true_pos = true_pos1 + true_pos2 + true_pos3
         false_pos = false_pos1 + false_pos2 + false_pos3
         false_neg = false_neg1 + false_neg2 + false_neg3

         weight = true_pos/(true_pos+false_neg+false_pos)
         print(weight)

         0.7585825027685493

In [25]: # The IoU for the dataset that never includes the hero is excluded from grading
         final_IoU = (iou1 + iou3)/2
         print(final_IoU)

         0.550946410259

In [26]: # And the final grade score is
         final_score = final_IoU * weight
         print(final_score)

         0.417938306786
```

Extracting the model and testing it on live simulator:

Downloaded the “config_model_weights” and “weights” file saved under the \data\weights directory and saved in local repository.

Followed these steps on our local system to watch the quad use your model to try and follow the target:

1. Copy your saved model to the weights directory `data/weights`.
2. Launch the simulator, select "Spawn People", and then click the "Follow Me" button.
3. Run the realtime follower script

Some snapshots of this:

Conclusion:

Since, this FCN architecture doesn't depend upon the size of the input image, the same model can also work well for following another object (dog, cat, car, etc.) instead of a human**. We just need to change the mask data and train the network again. We don't really need to change the network architecture whenever there is a change in the target or there is any change in the size of the input**. We may need to put a greater number of encoder decoder blocks to increase the accuracy of the model. The architecture can be enhanced by tuning hyper parameters further. We can also put few more layers and tune the filter sizes in each of the encoder and decoder blocks to make the model more robust.

Future Work:

We can do the following to improve the accuracy of the model:

a. Collect more data for training and validation from the simulator and train the architecture with a greater number of data sets. More distant training images will help in improving the model.
b. Tune the hyper parameters by looking at the predictions for each category of evaluation datasets.
c. Change the network architecture by increasing the number of layers, changing the filter depth and changing the activation function and see how the network behaves. It is assumed that increasing the depth of the layer would help in enhancing the accuracy of the model. Increasing the depth of the layer would be able to pick up more finer details at each pixel level.
d. Include pooling layers after each convolutional layer and see its effect on model accuracy