Problem Statement

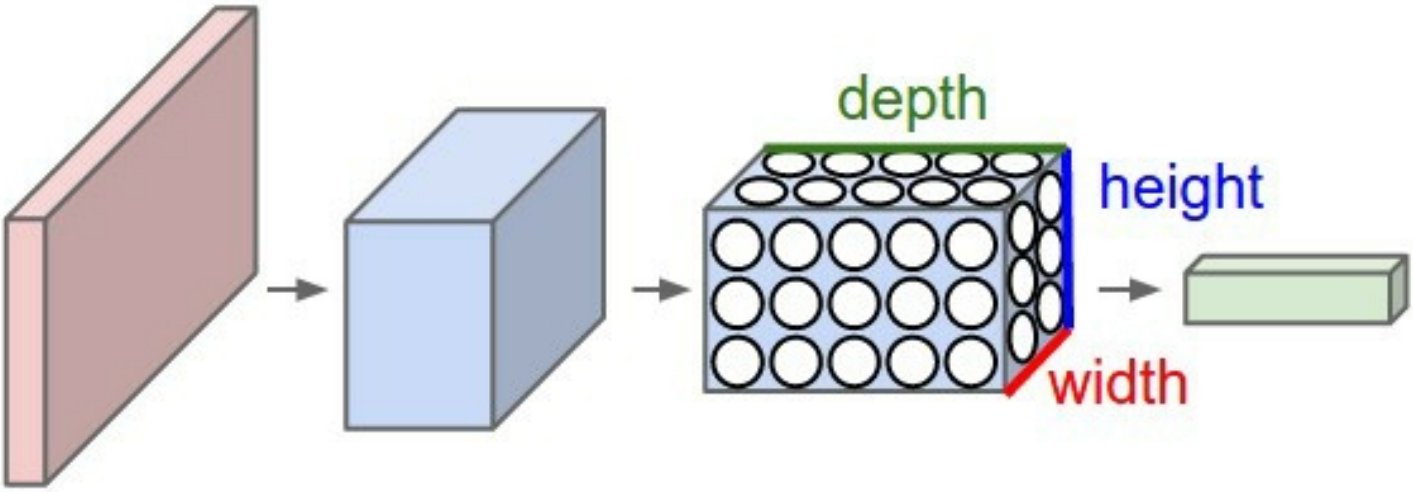
The objective of the project is to train a Fully Convolutional Network to identify and track a target in simulation. The weights trained in the model are used to track a person via drone in a unity engine simulation.

Content:

- Fully Convolutional Network
- Architecture
- Hyperparameters
- Evaluation
- Future Enchancements

Convolutional Neural Network

To understand FCN we first need to understand the CNN. In a regular deep neural network each node of the previous layer is connected to the node of the layer in front, this leads to good results when the numbers of weights to learn are small. But consider a case where the input is an image of 160x160x3, with a single hidden layer node the weights come around 76800. Clearly if we increase the size of the network, the model will overfit the data. To save us from the above dillema, comes a new technique called CNN.

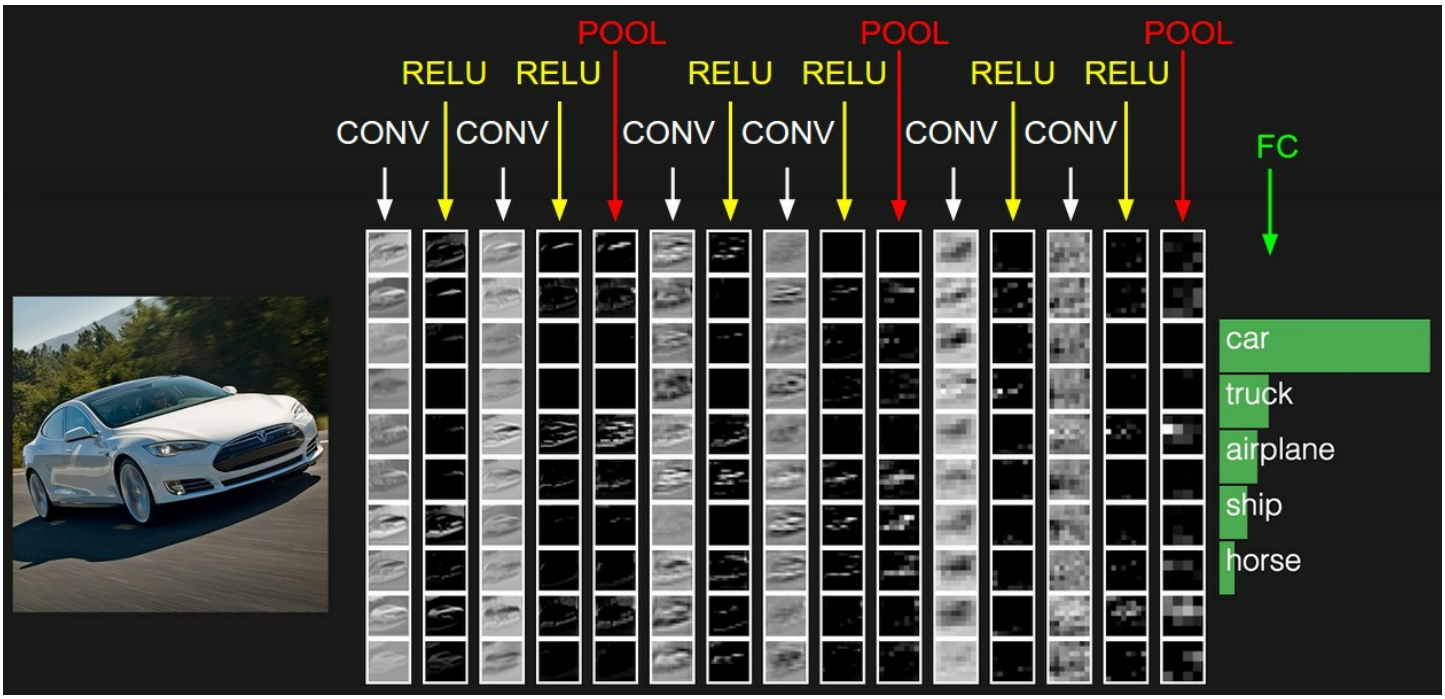


In CNN, the layer looks at a particular 3D volume of the previous instead of all its nodes. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. The final layer of a ConvNet is a fully Connected layer, which predicts the score of the classes to be predicted.

Here is guide to local connectivity in a CNN as per site http://cs231n.github.io/convolutional-networks/

> *Local Connectivity. When dealing with high-dimensional inputs such as images, as we saw above it is impractical to connect neurons to all neurons in the previous volume. Instead, we will connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the receptive field of the neuron (equivalently this is the filter size). The extent of the connectivity along the depth axis is always equal to the depth of the input volume. It is important to emphasize again this asymmetry in how we treat the spatial dimensions (width and height) and the depth dimension: The connections are local in space (along width and height), but always full along the entire depth of the input volume.*

The layers which are involved in a ConvNet are convolution, ReLU, Pooling, Fully Connected. The learnable parameters are in the convolution, Fully Connected layers as these consist of filters which transform the input image to the respective output. The ReLU and Pooling donot contain any learnable parameters.



Reference for images: http://cs231n.github.io/convolutional-networks/

FCN(Fully Convolutional Network)

When we want to segment an image instead of predicting a class, we use FCN or Fully Convolutional Network. Unlike the convolutional Neural network where the end layers are fully connected layer, in a fully convoluted network the fully connected layer are replaced by a 1x1 convolutional layer.

Network Architecture

FCN is primarily divided into two parts encoder and decoder block.

Encoder

Encoder block is similarly to the ConvNet. Steps followed in the encoder block are:

- Separable convolution
- Activation using ReLU
- Batch Normalization

Separable Convolution

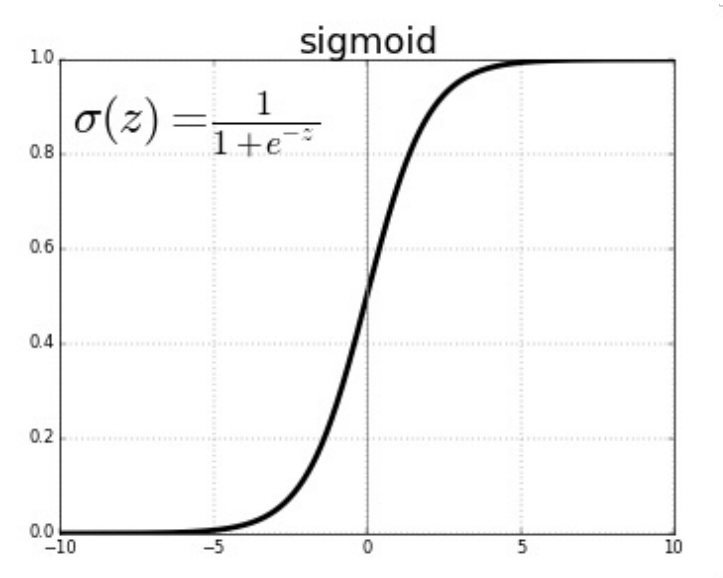
For the project, we have used separable Convolution instead of regular ConvNet to improve the performance by reducing the parameters in each convolution. In separable convolution instead of performing convolution of over the full depth of the input image, we perform convolution separate for each input channel and stack the result. After this we perform a depthwise 1x1 convolution across the channels.

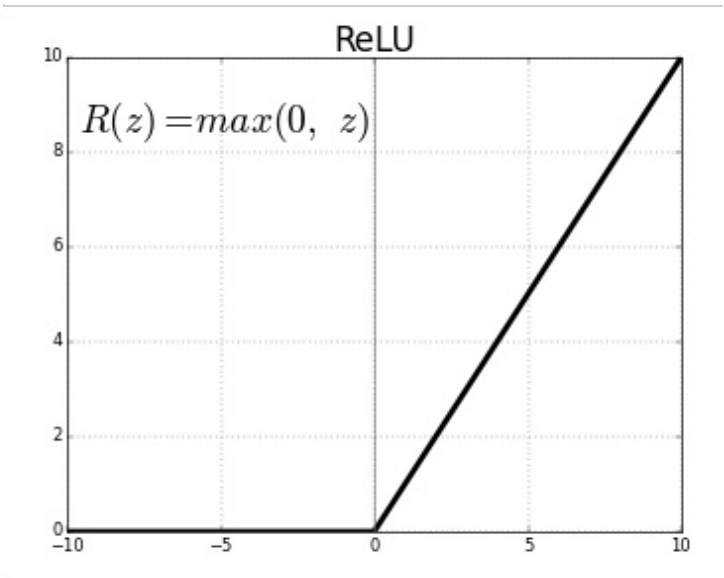
Reference: https://eli.thegreenplace.net/2018/depthwise-separable-convolutions-for-machine-learning/

```
from utils.separable_conv2d import SeparableConv2DKeras
separable_conv = SeparableConv2DKeras(filters, kernel_size, s
trides, padding, acivation)(input_layer)
```

ReLU (Rectified Linear Unit) Activation Function

Relu is an activation which is equal to the input if input is positive otherwise zero. This property helps tackle two problems: First, the problem of derivative of sigmoid function when the output approaches 1; Second, it helps to lower the no of parameters by making the negative term zero.





Reference: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

Batch Normalization

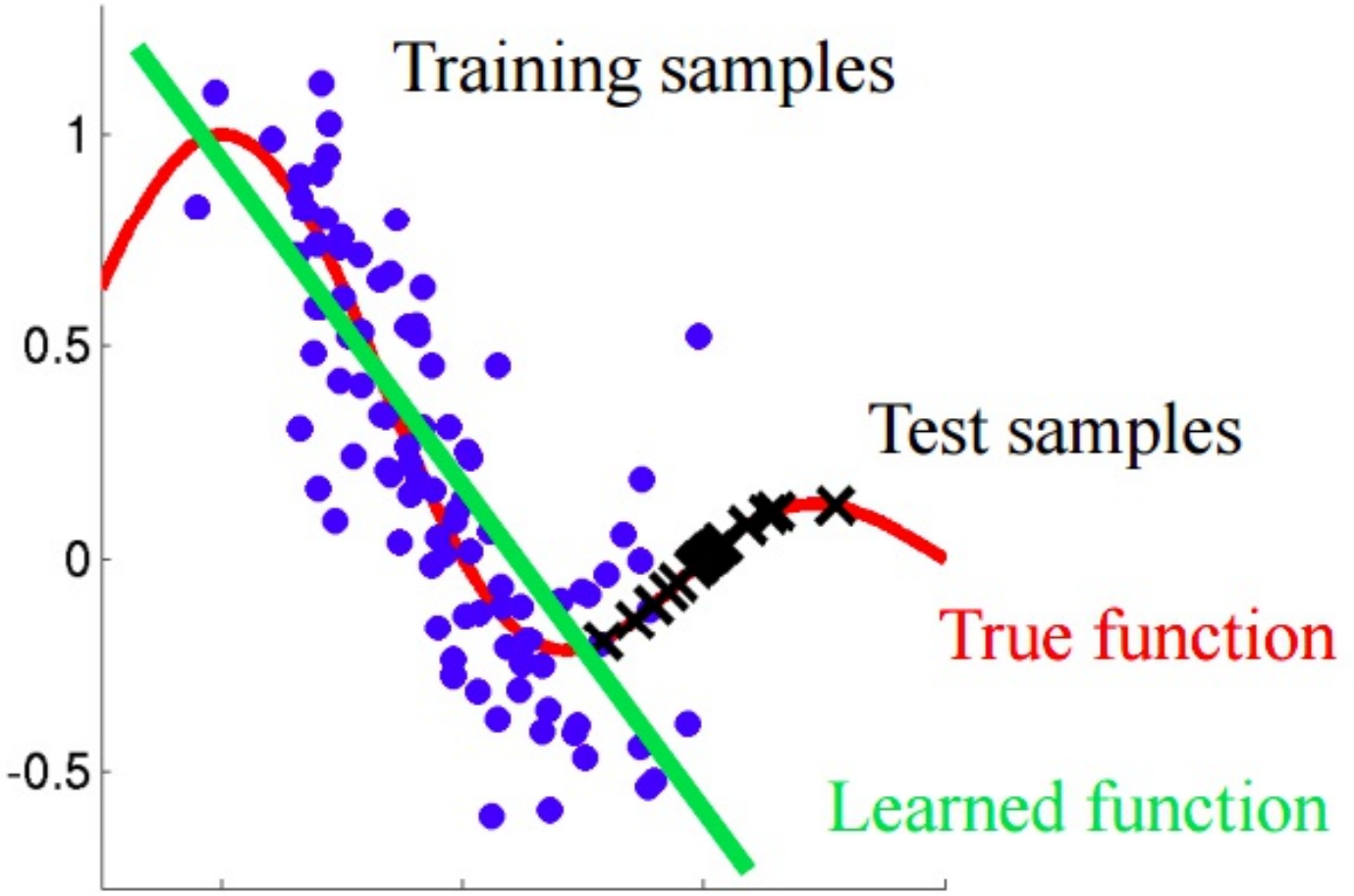
Batch Normalisation involves normalizing the output of each layer to a mean of 1 and standard deviation of 0. This is done in order to improve the stability of the Neural Network. Basically what it does is tackle the problem of internal Co Variate shift among layers of the network. So by doing batch normalization we normalize the distribution to make the network stable and learn faster. But this adds an extra step of computation within the network.

```
output_layer = layers.BatchNormalization()(input_layer)
```

Reference: https://arxiv.org/pdf/1502.03167.pdf

CoVariate Shift

CoVariate Shift refers to the change in distribution of data between the training samples and the test samples. This is depicted by the image below:



1x1 Convolutional Layer

1x1 Convolution means convolution of the previous layer by 1x1 2D filter, which is retains the 2D size of the layer on which the convolution is applied. The 1x1 convolution layer is simply a regular convolution, with a kernel and stride of 1.

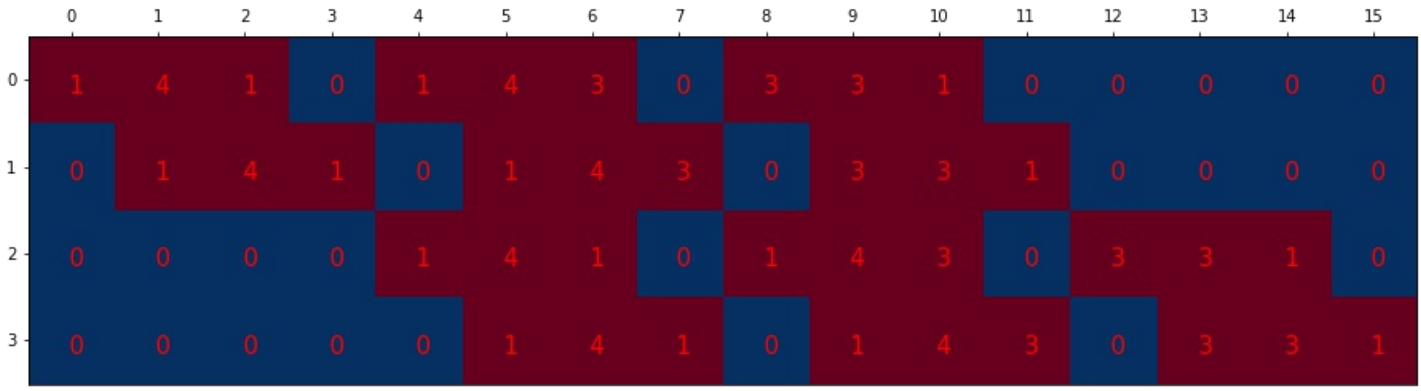
Decoder

The decoder block reforms the image/prediction by taking the output of the encoder block as its input. Decoder block usually consists of the following layer:

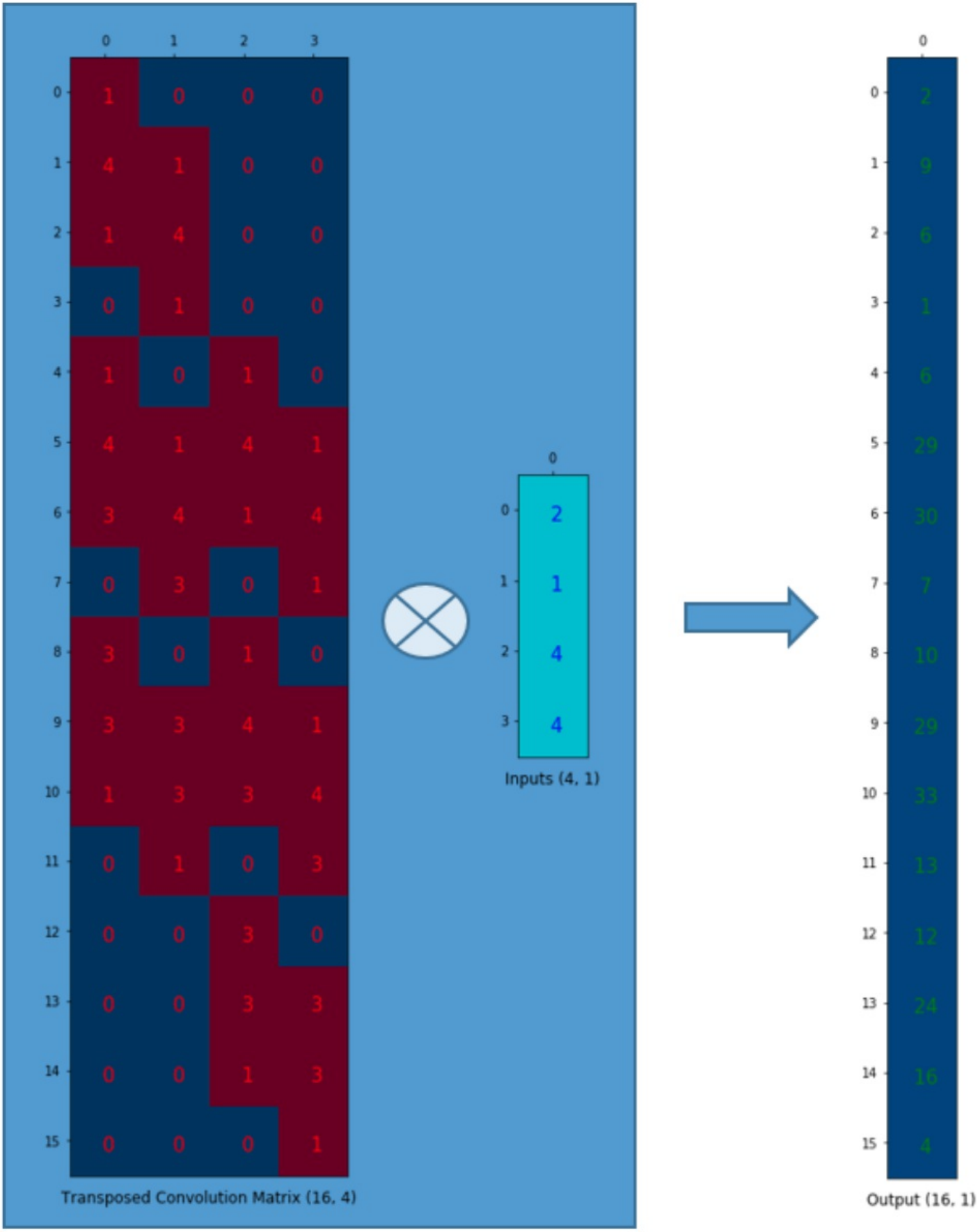
- Upsampling: Upsampling increases the resolution of the input image when applied. There are Bed of nails of upsampling where the input image data points are entered and all newly generated poistions are left zero. See image: There is also Nearest Neighbour unpooling which makes the nearest neighbour on the output image same as the data point on the input image which is the closest.

- Tranpose Convolution: This involves upsampling in a optimally, it differs from upsampling becasue the weights involved in the transformation are learnable through back propogation. To understand transpose convolution we first need to understand the convolution matrix. Let’s start from convolution, we have input matrix of 4x4 and we want to transform it to a 2x2 using a weight matrix of 3x3. To define the convolution matrix, we write the weight matrix as 4x16 as shwon in the image below, each row represents the convolution weights and they are strided by block of one from each other. After this, we flatten our input matrix i.e 4x4 to 16x1. Now, if we multiply the convolution matrix 4x16 to flattened input matrix 16x1 we get a column matrix of 4x1 which we can reaarange to get the 2x2 matrix. So we transformed the input from 16(4x4) to 4(2x2). In transposed convolution we want to do the opposite i.e.

transform from 4(2x2) to 16(4x4). So we take the transpose of convolution matrix and get a 16x4 matrix, again flatten the input matrix 2x2 to 4x1, and multiply the two to get a 16x1 column matrix. After rearranging we get the 4x4 matrix, which is desired. The weights of the convolution matrix can be learned through back prop therefore it is not a passive transformation like upsampling.



Convolution Matrix (4, 16)



Reference for images of Convolution Matrix and Transposed Convolution:

https://towardsdatascience.com/up-sampling-with-transposed-convolution-

9ae4f2df52d0

- Skip Connections: Skip connection connect one layer of the network to the other by skipping the layers in between them.

In the project the decoder block function consist of :

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

    return output_layer
```

Bilinear Upsampling

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.

```
def bilinear_upsample(input_layer,k = 2):
    output_layer = BilinearUpSampling2D((k,k))(input_layer)
    return output_layer
```

Concatenation

Concatenates two layer of same dimension 2D, implementation of skip connection in the network.

```
output_layer = layers.concatenate([output_layer, large_ip_layer])
```

Separable Convolution

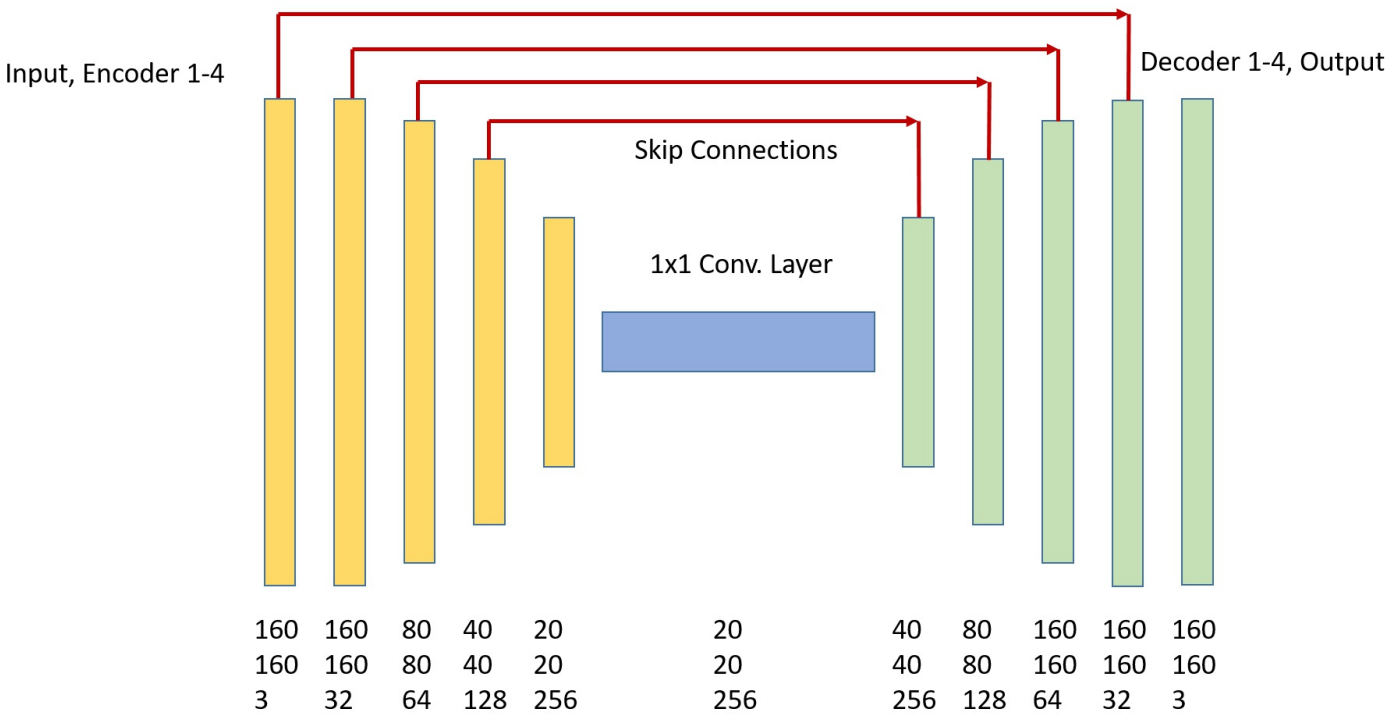
Refer to the section on Separable Convolution for detailed explaination.

```
output_layer = separable_conv2d_batchnorm(output_layer, filters)
```

Network Used

I used four encoder and decoder blocks. The input image was 160x160,I used a stride 1 in the first encoder layer as I didn’t wanted to decrease the spatial information. Afterwards I used the stride of two decreasing the spatial dimension till 20x20 then feeded the input to the 1x1 convolutional function, with a filter size of 256. Then I implemented 4 decoder to restore the image, with stride of 2 and used stride of 1 at the last layer. The

decoder block also included the skip connection in the for of layer concatenation.




```
def fcn_model(inputs, num_classes):

    # TODO Add Encoder Blocks.
    # Remember that with each encoder layer, the depth of you
r model (the number of filters) increases.
    encoder_1 = encoder_block(inputs, 32, 1) #160
    encoder_2 = encoder_block(encoder_1, 64, 2) #80
    encoder_3 = encoder_block(encoder_2, 128, 2) #40
    encoder_4 = encoder_block(encoder_3, 256, 2) #20

    # TODO Add 1x1 Convolution layer using conv2d_batchnorm()
.
    conv_1x1 = conv2d_batchnorm(encoder_4,256) # 20
```

```
    # TODO: Add the same number of Decoder Blocks as the numb
er of Encoder Blocks
    decoder_1 = decoder_block(conv_1x1,encoder_3,2,256) # 40
    decoder_2 = decoder_block(decoder_1,encoder_2,2,128) # 80
    decoder_3 = decoder_block(decoder_2,encoder_1,2,64) # 160
    decoder_4 = decoder_block(decoder_3,inputs,1,32) # 40

    x = decoder_4

    # 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')(x)
```

Hyperparameters

The hyperparameters

- batch_size: number of training samples/images that get propagated through the network in a single pass. I used a batch size of around 20. I kept the batch size low as I didn't wanted the system memory to overflow, either way I could compensate for this low number by increasing the steps per epochs.

- learning_rate: I set the learning rate at 0.01, this caused the model learn quickly but the coonvergence wasn't smooth. Decreasing the

learning rate to 0.001, smoothened the convergence but increased the time.

- num_epochs: number of times the entire training dataset gets propagated through the network. I set the number of epochs at 40. Increasing the number led to increase in training with diminishing increase in the accuracy/score.

- steps_per_epoch: number of batches of training images that go through the network in 1 epoch. I set the value at 250. As per the recommendation in the notebook, there were around 4200 images, therefore I took the number as 5000 and divided it by the batch size which gave me 250.

- validation_steps: number of batches of validation images that go through the network in 1 epoch. I set this at the default value of 50, I noticed that increasing this slighlty so increased the performance but definitely increased the time of computation.

- workers: maximum number of processes to spin up. I wanted to utilize the full performance of the px2large AWS instance therfore I set the value at 8.

Results and Evaluation

The results of the model on the sample evaluation are discussed in this section. The metric used for scoring here is the intersection over union metric. Below discussing the results lets talk in brief about the metric

intersection over union.

Evaluation

Evaluate your model! The following cells include several different scores to help you evaluate your model under the different conditions discussed during the Prediction step.

```
# 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 0.9963908016137142
average intersection over union for other people is 0.4053637087950692
average intersection over union for the hero is 0.9187339398982018
number true positives: 539, number false positives: 0, number false negatives: 0
```

```
# 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 0.9891523264715895
average intersection over union for other people is 0.7820823739540066
average intersection over union for the hero is 0.0
number true positives: 0, number false positives: 34, number false negatives: 0
```

```
# 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 0.9971015008501019
average intersection over union for other people is 0.4841400497362313
average intersection over union for the hero is 0.2070788478844215
number true positives: 120, number false positives: 1, number false negatives: 181
```

```
# 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)
```

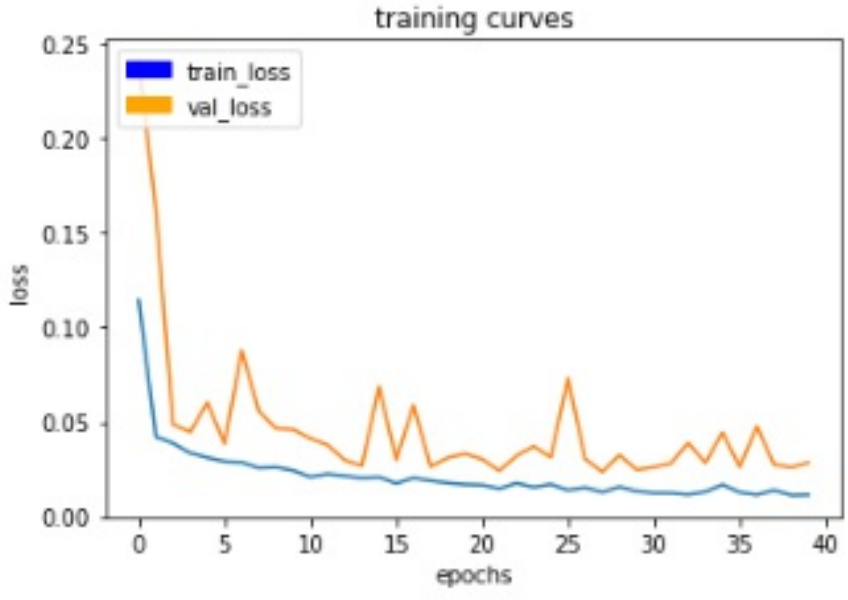
```
0.7531428571428571
```

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

```
0.562906393891
```

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

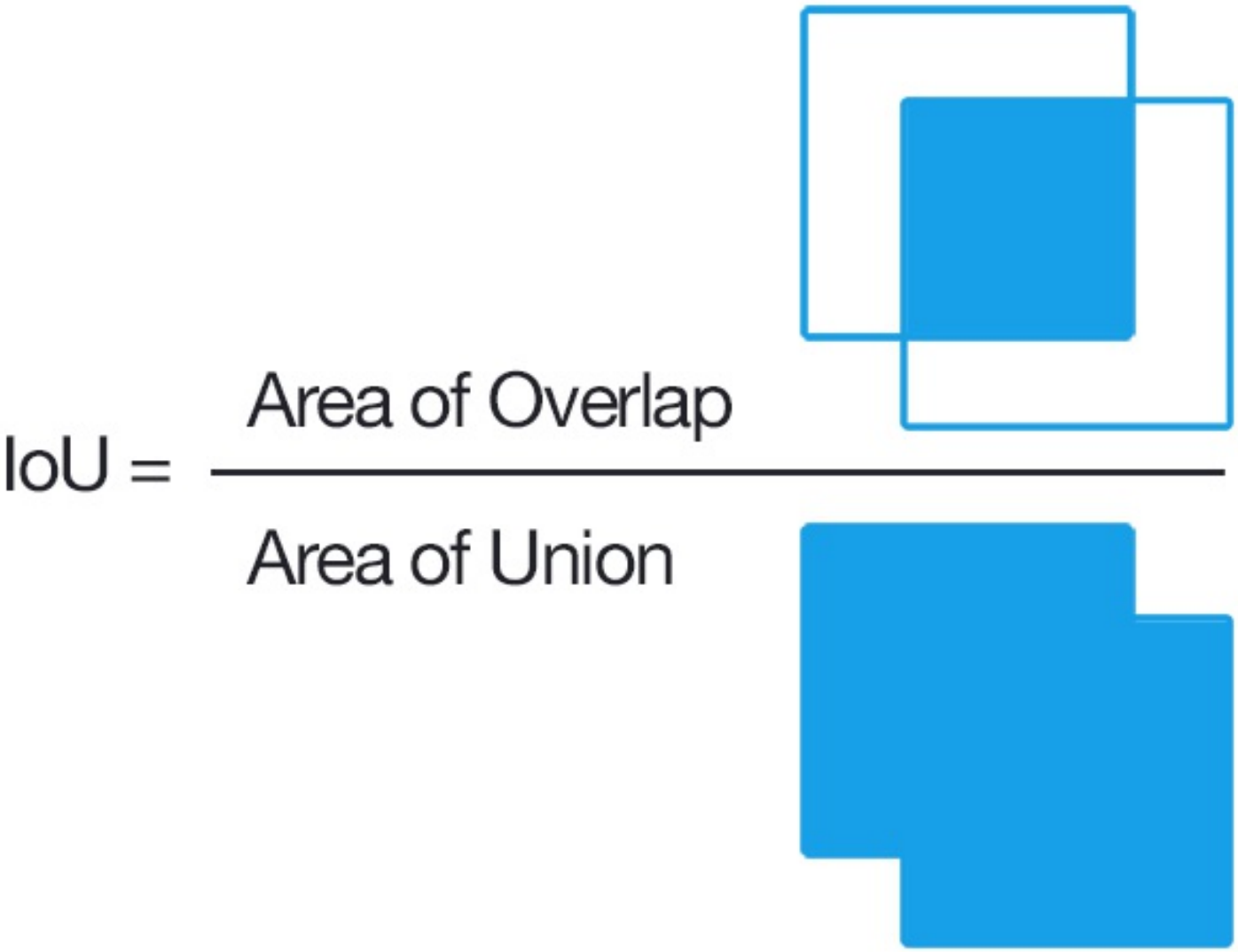
```
0.423948929799
```




```
250/250 [==============================] - 166s - loss: 0.0114 - val_loss: 0.0286
```

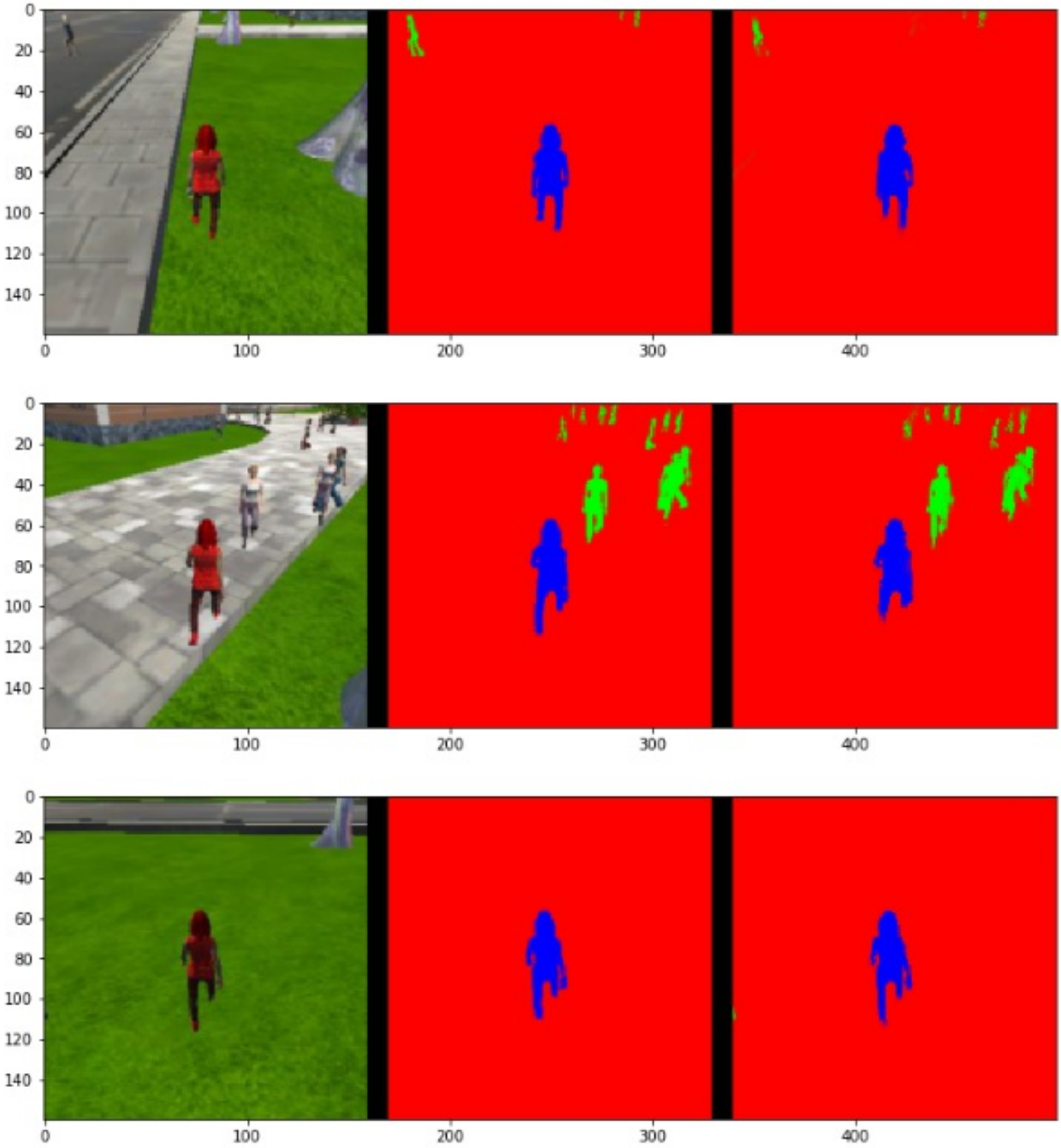
IOU - Intersection over Union

Intersection over Union or IOU is used to measure the accuracy of an object detector on a particular dataset. The formula describing the IOU is decrsribed in the figure below which the ratio of the area of the overlap by the area of the union. Area of over lap is the intersection area of the predicting box and truth box whereas the area of the union is the union of the both areas. As the ratio is of the intersection and union therefore the value is less or equal to 1 always. For the model I got the final score of 0.4239, with an IOU of 0.5629.

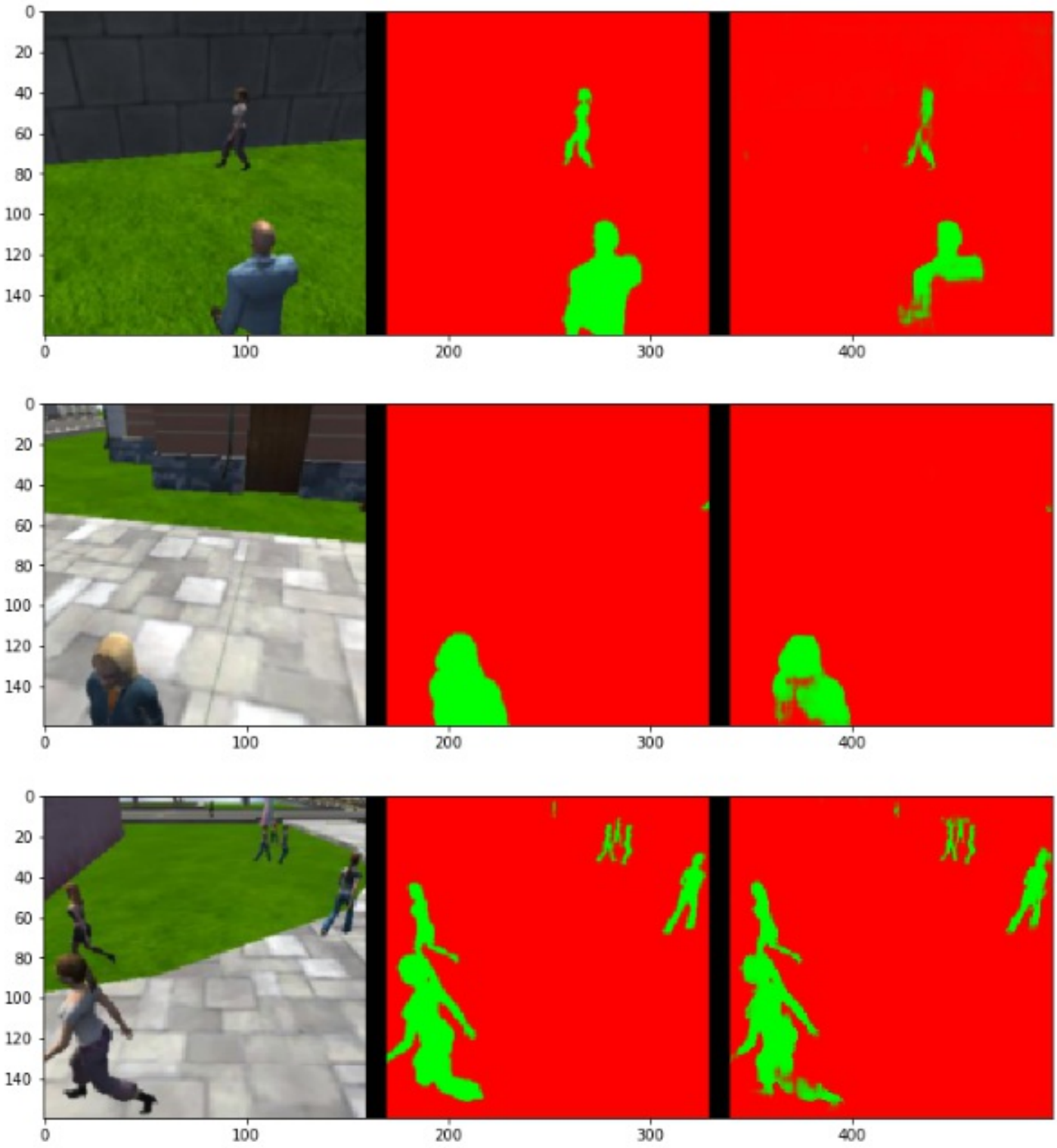


The results are as follows:

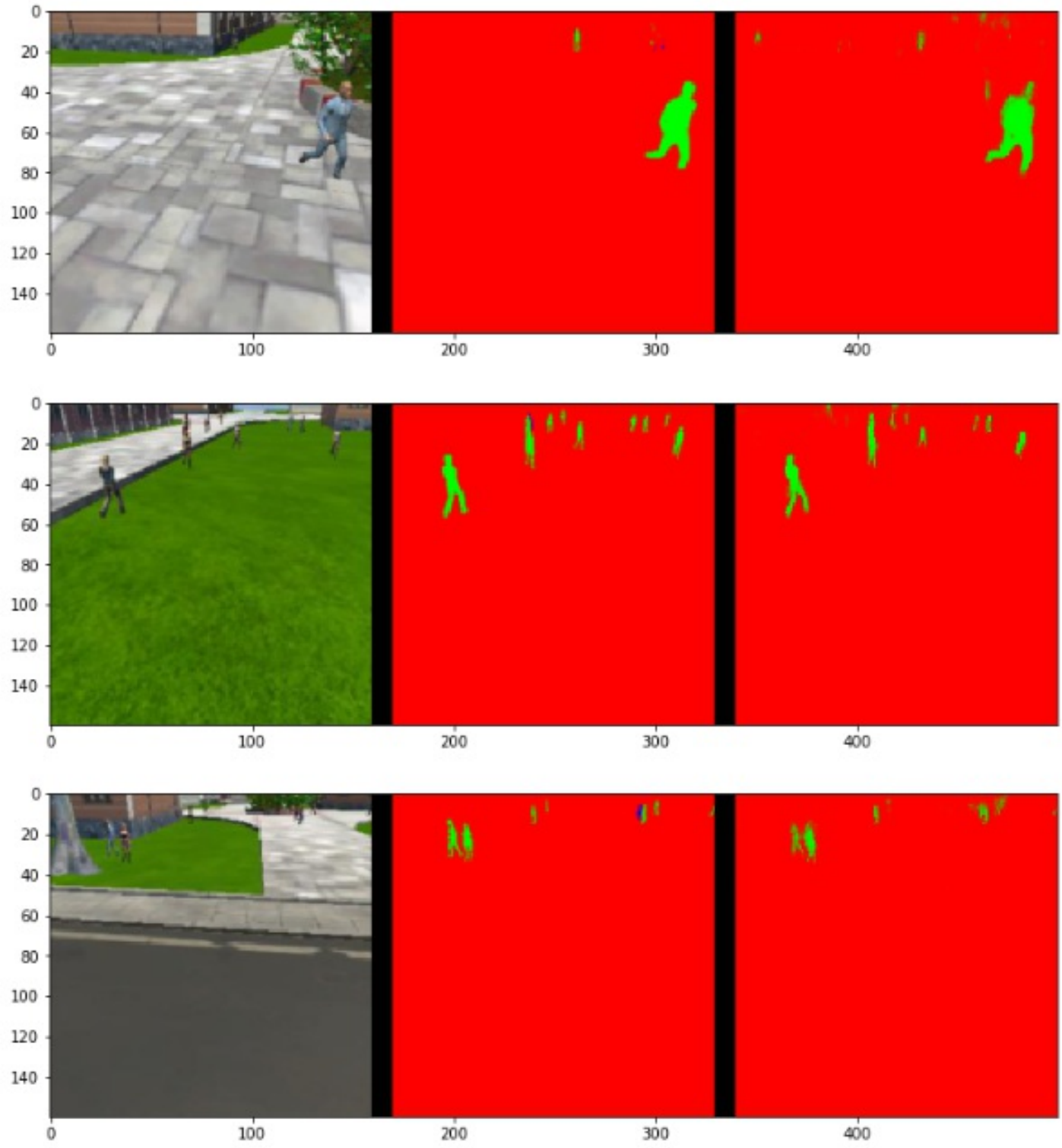
- Identifying the Target in follow mode.



- Identifying Non-Targets in patrol mode



- Identifying the Target in patrol mode



Future Enchancements

- I haven’t trained the model on my training set, therefore I would like to train the model on my training set, and see the results.
- The model can be trained for any object whether it is animal, bird

or other person on the condition that proper tarining set is provided. The unity engine environment provides the follow mode for only the person, therfore to use the same environment and train the model for following some other person, the unity game engine code has to be tweaked.