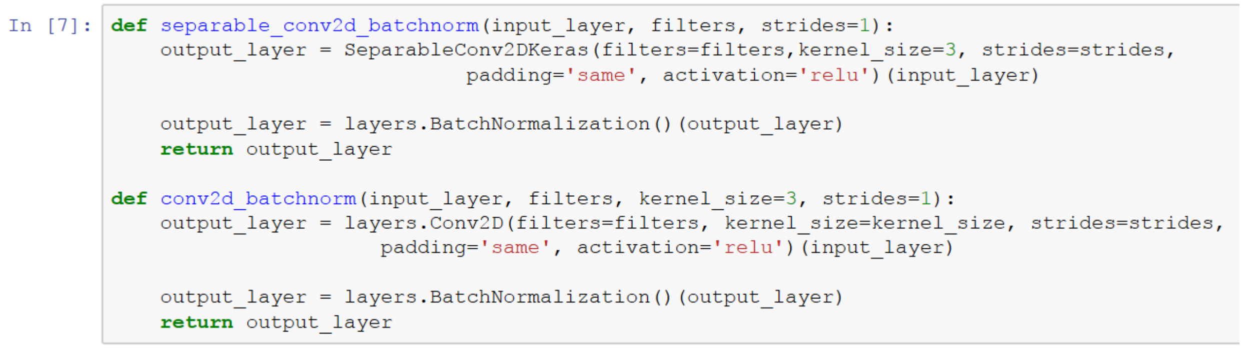
Arnold Kompaniyets

Robo ND (Term 1)
June 15, 2018

Follow Me
Write-Up

The Jupyter Notebook:

To start, I think it would be best to go step-by-step through each block of code in the Jupyter notebook for this particular project. Afterwards, specific parts of the coded neural network can be discussed in further detail.

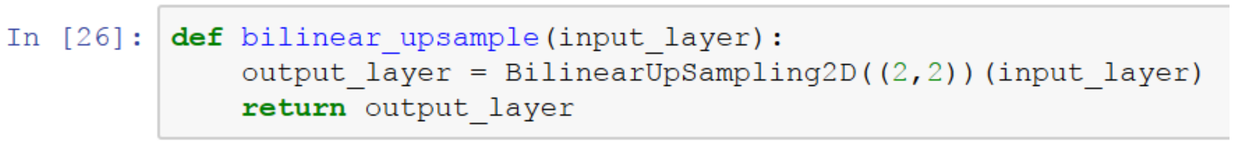


```
In [7]: def separable_conv2d_batchnorm(input_layer, filters, strides=1):
            output_layer = SeparableConv2DKeras(filters=filters,kernel_size=3, strides=strides,
                                 padding='same', activation='relu')(input_layer)

            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 first box of code defines two convolution functions to be used later, one for separable convolutions and another for a regular convolution. This code block isn't absolutely necessary, since Keras already does have inbuilt functions for both of these convolutions (SeparableConv2DKeras and Cond2D). However, having this code block does make implementing the convolutions far easier later on in the Jupyter notebook, simply in terms of writing less code. Of note here, the filter size is set as 3x3 pixels in both convolution functions, along with default strides set to 1 (meaning that the filter moves over one pixel after each convolution step). Padding was set to "same", meaning that enough outside 'padding' (simply a collection of zeros placed outside the image edges) is placed to have the resulting size of an image layer after a convolution the same, given a stride of 1 (or exactly half the size given a stride of 2, where the filter moves over two pixels after each convolution step).
- Furthermore, ReLUs are introduced after each convolution step, which adds an element of non-linearity to the data (although in terms of the math, the ReLU step simply sets all negative values to zero and keeps the positive values as is).
- Lastly, the values of each batch taken through a convolution step are normalized, which prevents inputs of high RGB values from having increased pull on the training

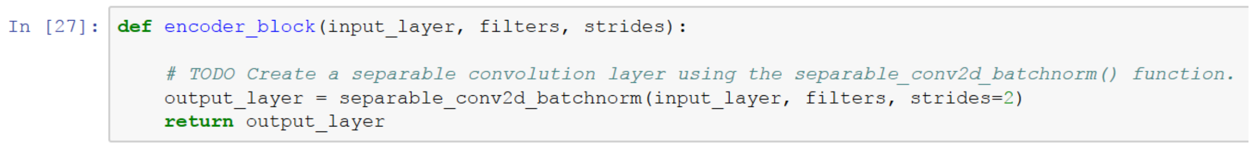
of a neural network. With all of this defined in this code block, the convolutions can be quickly implemented below.

- As mentioned in the course lessons, the main difference between the two types of convolutions is the reduction in needed parameters in separable convolutions. In a separable convolution, each input layer is initially only traversed by one kernel, providing for a feature map for each layer. These feature maps are then traversed by the desired number of kernels (as 1x1 convolutions). This allows for less total parameters that result from each convolution, which reduces runtime and can to a certain extent reduce overfitting (since a neural network “learns” much quicker with a greater number of parameters).



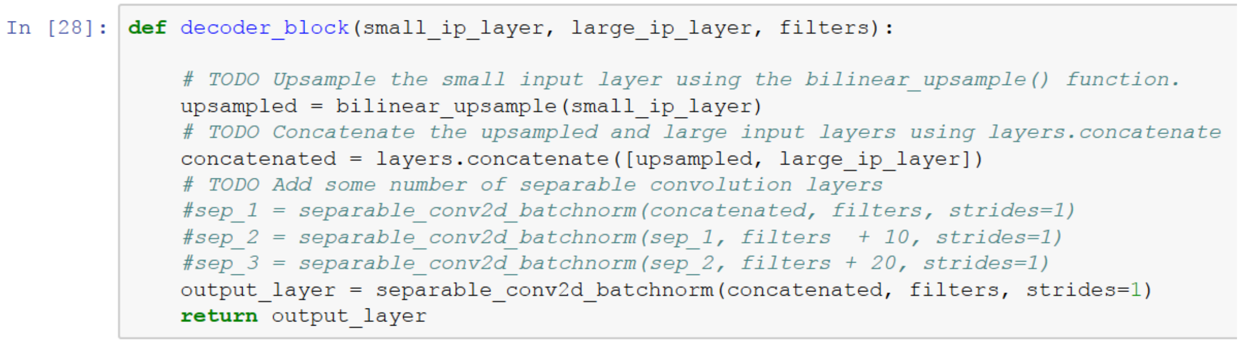
```
In [26]: def bilinear_upsample(input_layer):
             output_layer = BilinearUpSampling2D((2,2))(input_layer)
             return output_layer
```

- This block of code performs a bilinear upsampling of image data. With the default values for the upsampling set to (2,2), this means that a 2x2 image would be upscaled to a 4x4; basically, the upsampling doubles the size of the image. This is done by means of mathematic approximation: the points whose exact values are known are, in essence, pulled apart, and the in-between unknown values are approximated on a gradient. This, of course, is not perfectly precise, but is an effective way of bringing images up to a desired size.



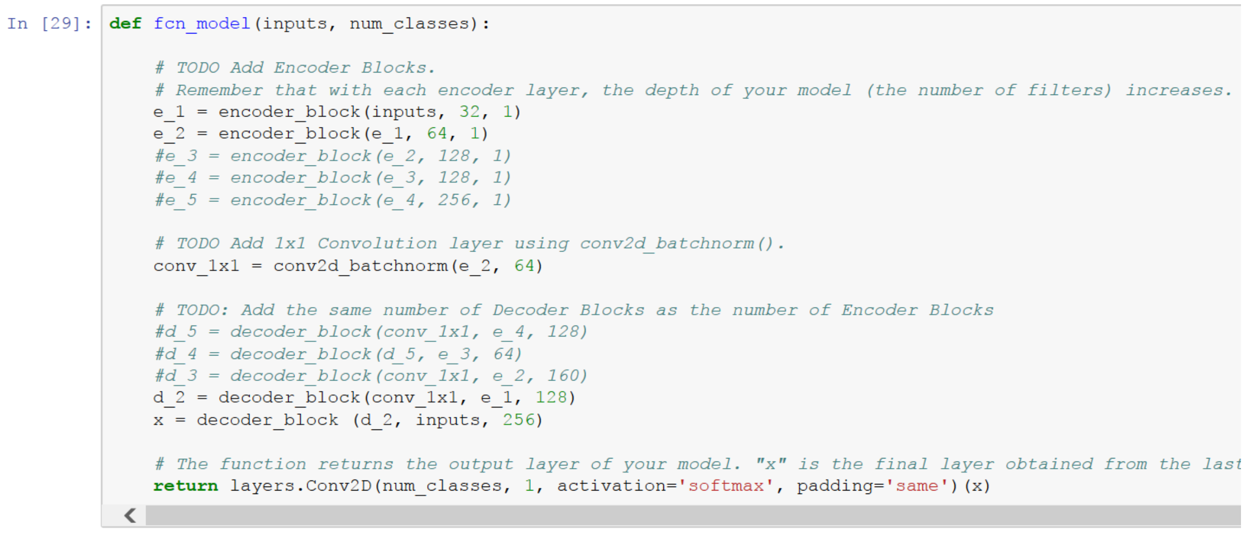
```
In [27]: def encoder_block(input_layer, filters, strides):

             # TODO Create a separable convolution layer using the separable_conv2d_batchnorm() function.
             output_layer = separable_conv2d_batchnorm(input_layer, filters, strides=2)
             return output_layer
```

- The encoder block is simply invoking a single instant of the separable convolution function. A point of interest, however, is that strides were set to ‘2’. This was done because the upsampling layer, by default, doubles an input layer’s size; as such, the encoder needs to halve the original layer size with each step in order for subsequent steps to have matching layer sizes.



```
In [28]: def decoder_block(small_ip_layer, large_ip_layer, filters):

             # TODO Upsample the small input layer using the bilinear_upsample() function.
             upsampled = bilinear_upsample(small_ip_layer)
             # TODO Concatenate the upsampled and large input layers using layers.concatenate
             concatenated = layers.concatenate([upsampled, large_ip_layer])
             # TODO Add some number of separable convolution layers
             #sep_1 = separable_conv2d_batchnorm(concatenated, filters, strides=1)
             #sep_2 = separable_conv2d_batchnorm(sep_1, filters  + 10, strides=1)
             #sep_3 = separable_conv2d_batchnorm(sep_2, filters + 20, strides=1)
             output_layer = separable_conv2d_batchnorm(concatenated, filters, strides=1)
             return output_layer
```

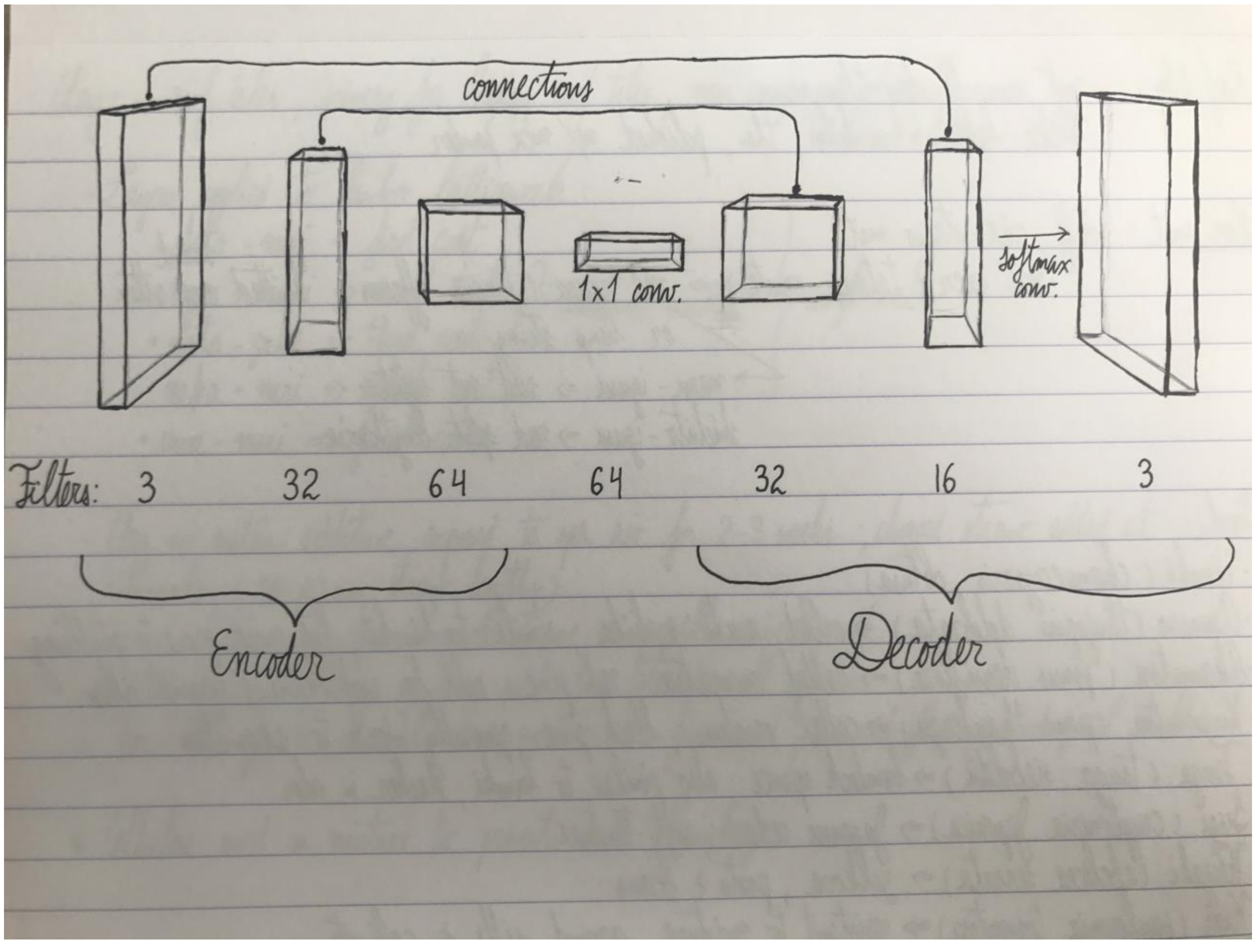
- The decoder block is slightly more complex. First, the input layer is upsampled using the technique described above. Afterwards, this upsampled layer is concatenated with a layer of identical size from the encoder blocks. This is done to make 'connected layers', a topic that was discussed in the course lessons to help provide better spatial resolution of the final network layers.
- Following this, a separable convolution is performed on the concatenated layer to complete the decoder block. It is visible from the image above that multiple convolutions were also attempted in the network training process, although in the end, simply having one proved best. This will be discussed in further detail later.



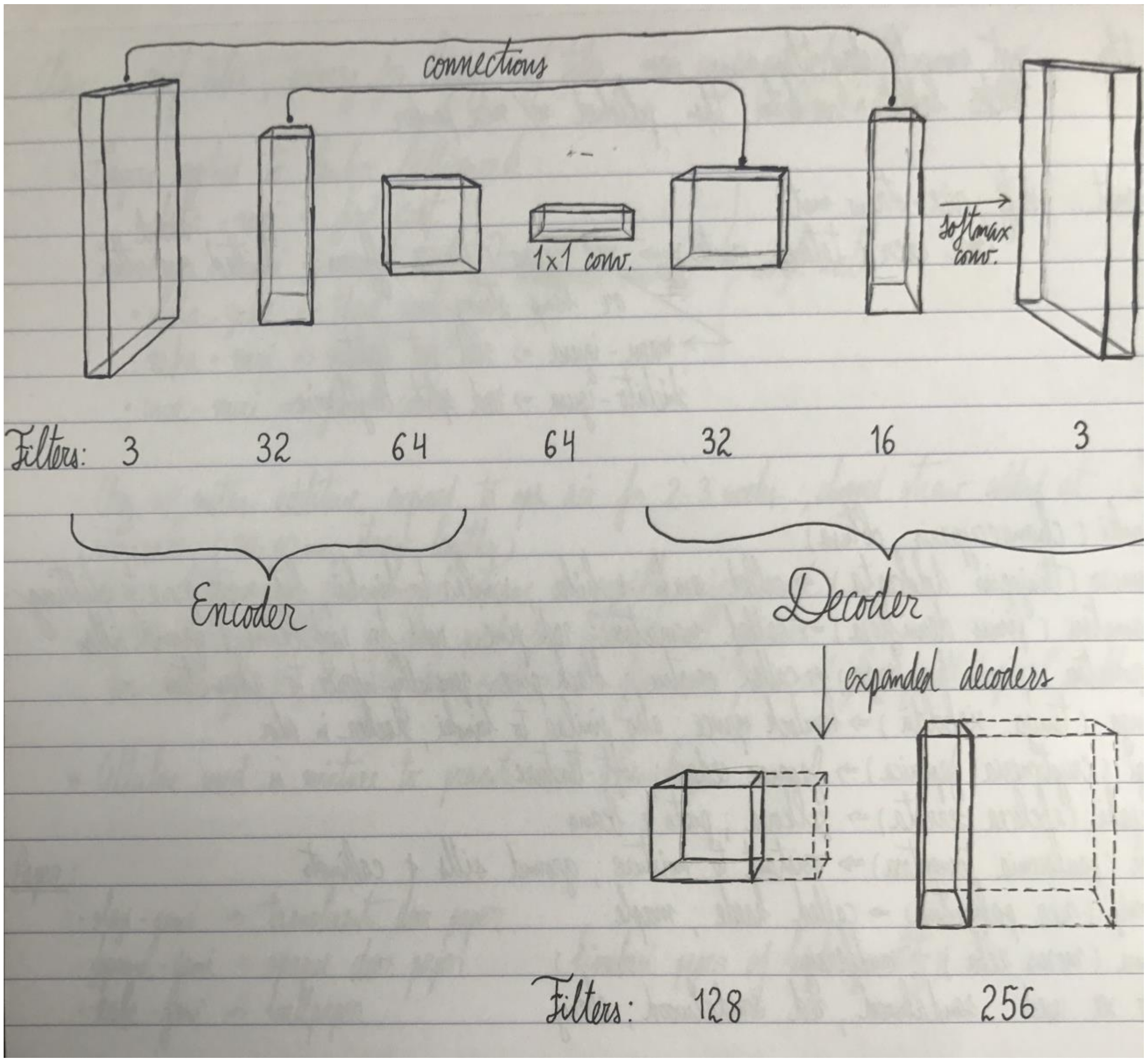
```
In [29]: def fcn_model(inputs, num_classes):

             # TODO Add Encoder Blocks.
             # Remember that with each encoder layer, the depth of your model (the number of filters) increases.
             e_1 = encoder_block(inputs, 32, 1)
             e_2 = encoder_block(e_1, 64, 1)
             #e_3 = encoder_block(e_2, 128, 1)
             #e_4 = encoder_block(e_3, 128, 1)
             #e_5 = encoder_block(e_4, 256, 1)

             # TODO Add 1x1 Convolution layer using conv2d_batchnorm().
             conv_1x1 = conv2d_batchnorm(e_2, 64)

             # TODO: Add the same number of Decoder Blocks as the number of Encoder Blocks
             #d_5 = decoder_block(conv_1x1, e_4, 128)
             #d_4 = decoder_block(d_5, e_3, 64)
             #d_3 = decoder_block(conv_1x1, e_2, 160)
             d_2 = decoder_block(conv_1x1, e_1, 128)
             x = decoder_block (d_2, inputs, 256)

             # The function returns the output layer of your model. "x" is the final layer obtained from the last
             return layers.Conv2D(num_classes, 1, activation='softmax', padding='same')(x)
```

- This block of code puts together the pieces discussed above into a complete model for neural network training. First, a series of encoder blocks are placed. Each block halves the size of the image but adds many more layers of depth than the initial 3 layers (the RGB layers). In the commented-out code, it can be seen that up to five encoder blocks were attempted during the training process, but eventually it was found that two were sufficient to achieve the desired task. The reasoning for why will be discussed in more detail later. Additionally, it can be seen that the chosen filters for the encoder blocks were 32, followed by 64. Both less and more filters were tried, with reasoning for the final choice also discussed later.

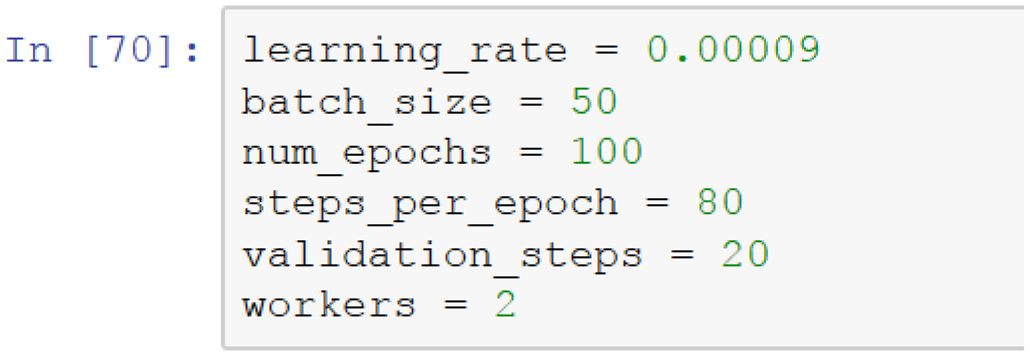
- Following the encoder blocks, a 1x1 convolution is done on the final encoder output. This is simply a normal convolution with an identical number of filters, in this case 64. This is normally the place where a fully connected layer would be formed when making a connected neural network. The difference in choice will be discussed further below.
- Continuing, the decoder blocks are written in. As mentioned earlier, up to five layers were tried, but in the end two were decided to be best. Therefore, two decoder blocks coded in. With the course lessons, it was stated that the typical structure of a neural network increases the filter number and decreases the layer size during encoding, keeps the size identical during the 1x1 convolution, and then follows to increase the layer size and decrease the filter number during decoding. The complete network, with included connected layers, would look as such:



- However, after much testing (approximately over the course of one month), it was found that a superior score was attained by continuing to increase the filter numbers through the decoder steps, until the final softmax function filter reduction of depth back to 3, visualized as such:



- My reasoning for why this worked is as follows. In looking at the decoder block code, it is essentially just the encoder block, the difference being that the convolution uses a stride of 1, with the input layer upsampled and connected to a previous layer prior to the convolution. This actually should produce a superior convolution, because the layers used are larger, connected to other layers, and a stride of 1 is used. As such, there is no reason to not continue adding additional filters as a means to further extract higher level features. The final softmax function still brings all of this data back into a 3 layer image.



```
In [70]: learning_rate = 0.00009
         batch_size = 50
         num_epochs = 100
         steps_per_epoch = 80
         validation_steps = 20
         workers = 2
```

- The last step before initiating training is setting the proper parameters. This took an extensive amount of tweaking and will be discussed in detail a bit later. In the final model, the learning rate was set to '0.00009'. This parameter governs how rapidly the model adjusts its weights towards the "correct" direction, as directed by the correctly labeled mask images. A value of 1 would lead to far too rapid changes in weight values from image to image, making it virtually impossible for the neural network to extract common features between the images. However, a very slow learning rate could lead to too many passes of the same image set through the neural network, leading to overfitting and network image memorization before enough feature data is extracted. I was initially quite reticent in lowering learning rates, because I was very worried about overfitting. However, after extensive training, I saw that overfitting the model was actually quite difficult, making me feel more comfortable lowering the learning rate. Since increasing filter number also increases the pace of neural network learning, the learning rate had to be lowered to a very low value when filter numbers reached values of 128 and 256.
- The batch size simply determines how many images are analyzed with each step in an epoch. With more images to analyze in a given step, the network can draw out superior features and connections. However, this does also lead to greater computational demands. As such, the batch size was set purely to the limit of the AWS instance. The batch size was started at 150 and moved down until an error message to allocation limit was not reached. 50 was approximately at that limit.
- The number of epochs was set to 100 in this code block, although this simply states how many epochs are gone through before the code block stops on its own and allows the neural network model to be saved. In reality, as will be discussed below, many more epochs are performed before the model is appropriately trained.
- The steps per epoch and validation steps determine how many batches of each image set are processed within each epoch. The general suggestion of keeping this to approximately the number of total images was maintained. With the training set

being about 4,000 images and the validation set 1,000 images, 80 and 20 steps were used, respectively (with a batch size of 50).

- The number of workers was also set to the default size of 2. This particular aspect of the code was not discussed in any detail during the course lessons, so it was decided to best not alter this perimeter.

Training the Network:

What a journey this turned out to be! After doing the semantic segmentation lab earlier in the course, I was able to establish a model of similar structure and achieve what I thought were good results. Using three encoder-decoder layers, I initially set the filters to 16-32-64-64-32-16-8. I figured using a learning rate of 0.01 was plenty sufficient and went on to train my neural network. With a batch size of 50, I initially went through around 50 epochs. This resulted in a training curve that dropped very abruptly and quickly settled around 0.1 loss, within two or three epochs. The rest of the epochs resulted in slow drops in training loss, with validation loss settling around 0.05. After 50 or so epochs, the training loss was around 0.03 and steadily dropping. The validation loss, though, began rapidly hovering between 0.03 and 0.06, high one epoch, low the next.

In evaluating this first run, I was able to achieve a final score of around 30%. This made me quite optimistic, because I thought I was relatively close to success. However, this couldn't be further from the truth. Initially, I thought that perhaps simply training the model more would result in a superior score. After hundreds more epochs, the highest final score I could achieve was around 33%. Past a certain point, doing more epochs could even cause a slight drop in score to 31% or 32%. Clearly, this was not sufficient.

As such, I began to change parameters. Increasing the batch size appeared to help initially. With a batch size of 100, I was able to achieve final scores of around 34%, but not higher. Increasing the batch size further would lead to much longer runtimes, yet the final score would not budge past 34%.

At this point, I went back to the original batch size and began to tweak the learning rate. Given the very rapid descent of my learning curve, I realized that I was training the model too quickly. I lowered the learning rate to 0.001 and observed the results. The training curve now exhibited a much better and more fluid downward curve. This optimism turned sour, however, once I saw that this fluid curve continued only until 0.01 validation loss. At this point, the validation curve once again started fluctuating wildly, eventually ending up cycling between 0.03 and 0.06 loss. Unfortunately, I was once again only able to achieve a final score of 34%.

Lowering the learning rate further continued to decrease the rapid descent of the initial phase of the training curve. However, after much waiting, the validation curve once again began to rapidly fluctuate between 0.03 and 0.06 loss, even while the training curve continued to decrease fluidly. This resulted in no improvement in my final score, even while I was able to achieve training curve losses at levels I had not achieved before, such as 0.02 or even 0.015.

At this stage, with no other parameters to tweak, I began to look at my actual model. In reading about neural networks, I found that each deeper layer continues to extract higher level features and information, so I thought perhaps the three layers of my current model were not sufficient. Thus, I added one more layer, all to see no improvement in my final score. I spent days once again tweaking the various parameter with my new model – batch sizes, learning rates, etc. Once again, 33 or 34% was the best I could achieve.

Beginning to grow despondent, I went to five layers. Once again spending multiple days training networks with all types of different parameters, I actually began to see worse final scores, this time only being able to achieve scores around 26 or 27%.

Feeling all else had already been tried, I decided to start altering the number of filters used. Believing the basic logic that more filters would lead to greater data extraction, I began to start doubling filter numbers (to 64, or 128, or 256). Run times certainly increased, but not the final score. I even decided to go as far as I could, trying to set the filter numbers to 1000 or 2000, and simply began to seen memory allocation errors. The higher number of filters resulted in the training loss rates dropping much faster (forcing me to further decrease the learning rate), but the dreaded leveling-out of the validation loss curve still occurred. With large filter numbers and five layers, I was able to still achieve a fluid training loss curve, reaching loss rates of around 0.004, yet the validation loss rates would still fluctuate around 0.03-0.06. The final scores were not improved neither, still being around 26 or 27%.

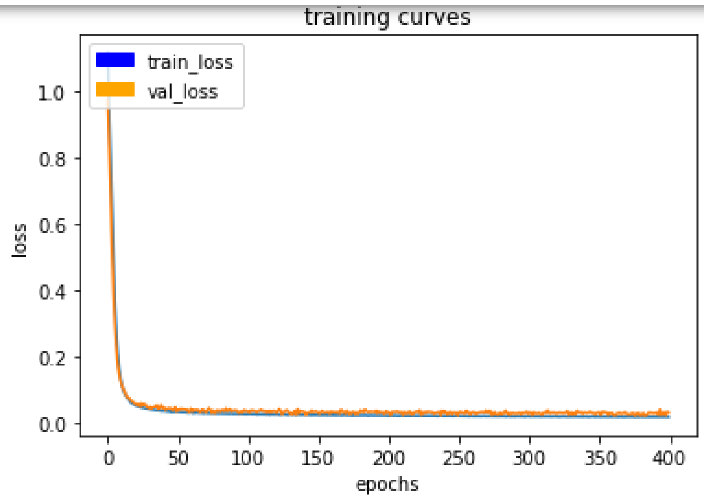
For the sake of experimentation, I tried using very low filter numbers as well, once again with all types of different parameter values. This kept the learning rate of the model to a decent pace, but did not result in any higher final scores.

In reading further about neural networks and extensively analyzing the Slack forums, I saw that this project itself may have intrinsic structural troubles, particularly relating to the enforced sizes of images used. By having a required block of code reduce each image to a size of 160x160, halving the feature map size every neural network layer would lead the feature map to be incredibly small by the time four or five layers are introduced. As such, it appears that far too much spatial information is lost at that point and prevents the neural network from achieving a high enough score.

At this point, I realized that while in an ideal scenario, building the more complex neural network, containing plethora filters, would produce a better final score, this particular project would require some adjustment. First, I decided to try and maintain better spatial resolution by preventing the encoder block from reducing image size. After a bit of experimentation, I found that a way to accomplish this was to set the bilinear upsampling to (1,1) and lowering the encoder stride to 1. This accomplished the desired task, yet once again, all that resulted was incredibly longer run-times. The final score remained capped around 34%, even though the image resolution remained at 160x160 throughout each neural network layer. Perhaps, this resolution was already relatively low to begin with and maintaining it was insufficient.

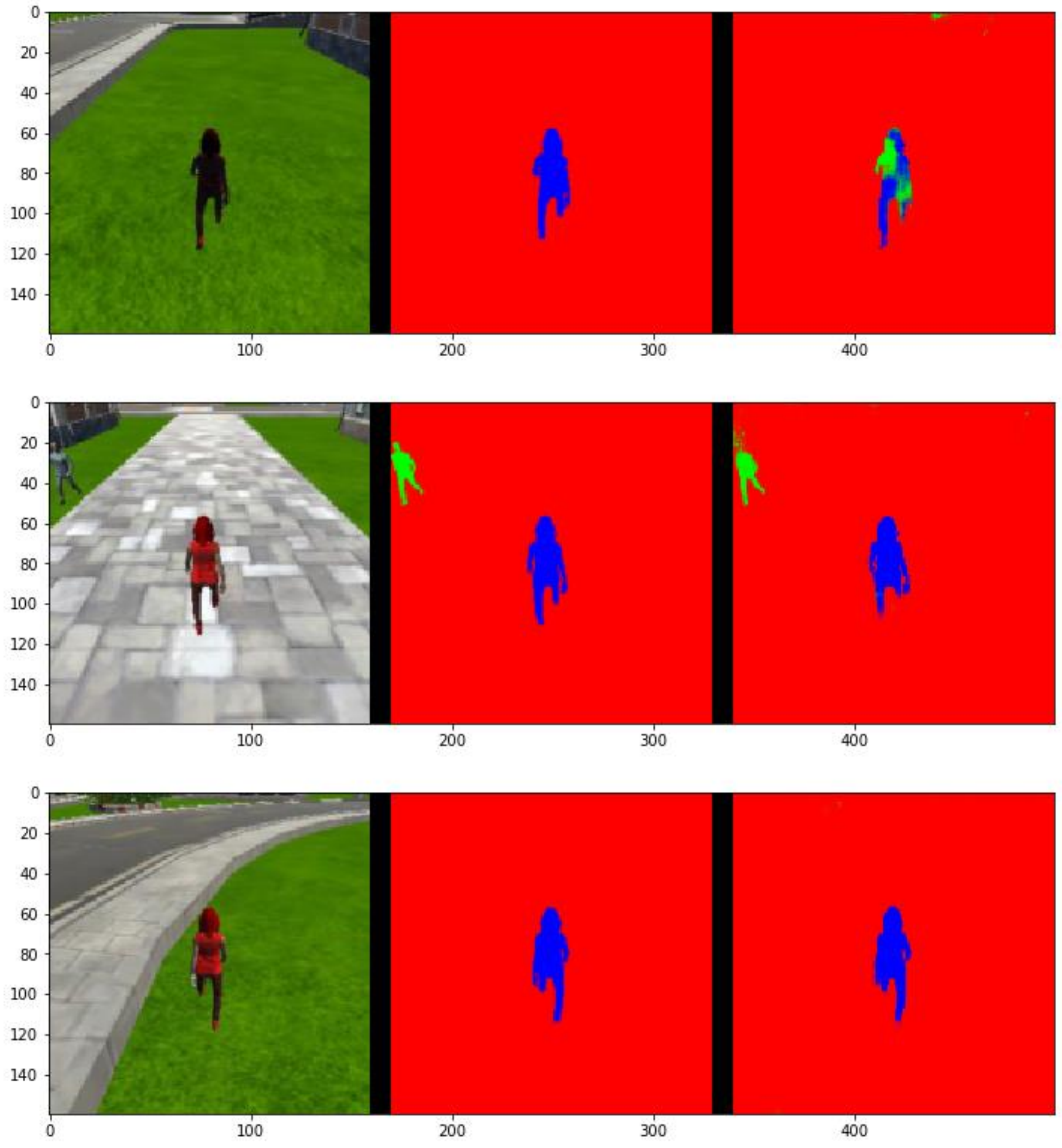
After this, the only other option I had was to start reducing the number of layers. Since I knew three encoder-decoder layers would not lead me to a sufficient final score, I decided to simply go to the extreme and reduce my model to one layer. Once again, I spent a few days altering all the parameters I could: filter number, learning rate, batch number, etc. Initially, my final scores were around 30%, but after continual training (hundreds more epochs), the final score once again moved up to a cap of around 34%.

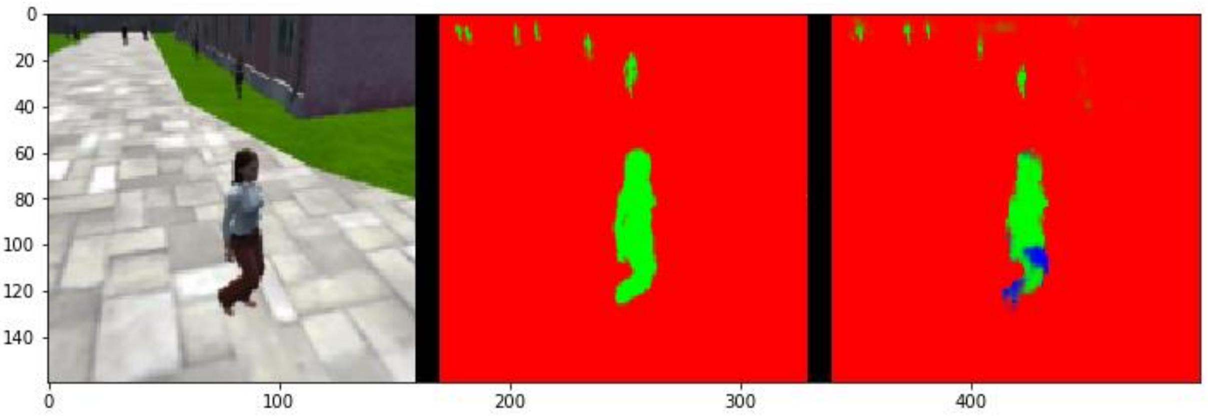
Feeling excruciating frustration at this point, I decided to just try a model of two layers anyway and see if by chance I could achieve a final score of higher than 34%. To my surprise, after a day of experimentation, I was able to discover parameters sufficient to achieve 35%. With 32-64 filters and a very low learning rate (around 0.0005), a few hundred more epochs was able to produce a final score of 36%. This produced both a tremendous feeling of success but also a bit of irritation, because I began to feel that after multiple weeks of work, I had simply been playing a guessing game, and only after abandoning my sense of logic (and choosing to just optimize the python code itself), I began to see some better results. Nonetheless, I had to continue with this, since I had finally come closer to the magical 40% than ever before. Hundreds of epochs later, perhaps even 1000 more, and I was able to achieve a final score of 37%. This was now quite promising, and I had to figure out what parameters would lead bump me up the necessary 3%. Adding more and more filters didn't quite seem to do the trick. It did require a much lower learning rate, but just didn't quite increase my final score. Eventually, after spending much time thinking about the actual structure of the encoder versus decoder block (as mentioned above), I decided to keep increasing the filter number through the decoder blocks. Using this "decoder expansion" technique, I was finally able to continue raising my scores. After approximately 400 epochs at the settings described previously, I achieved the following training curve:

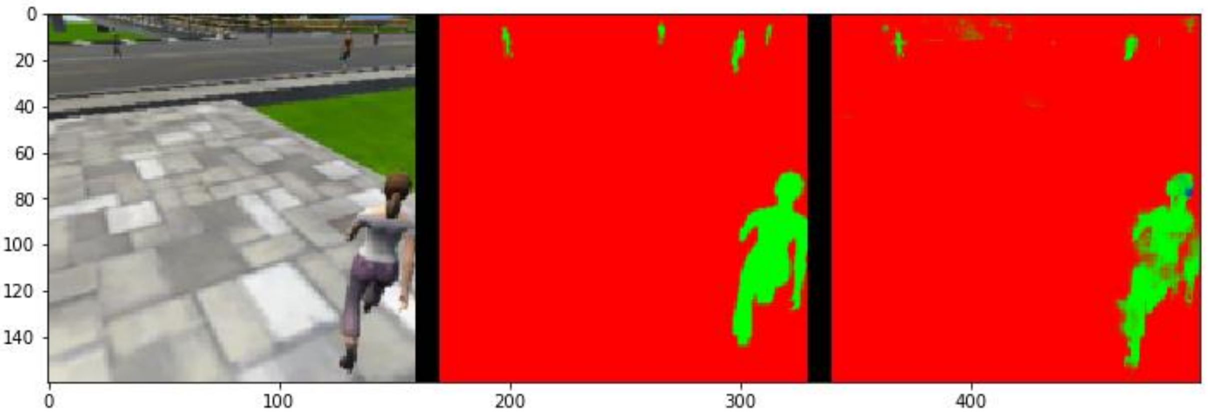


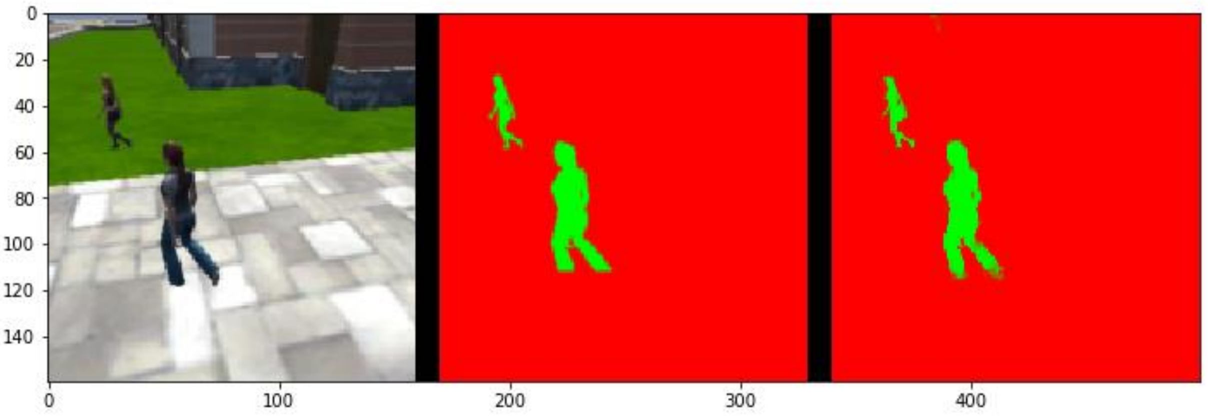
```
80/80 [==============================] - 110s - loss: 0.0169 - val_loss: 0.0305
```

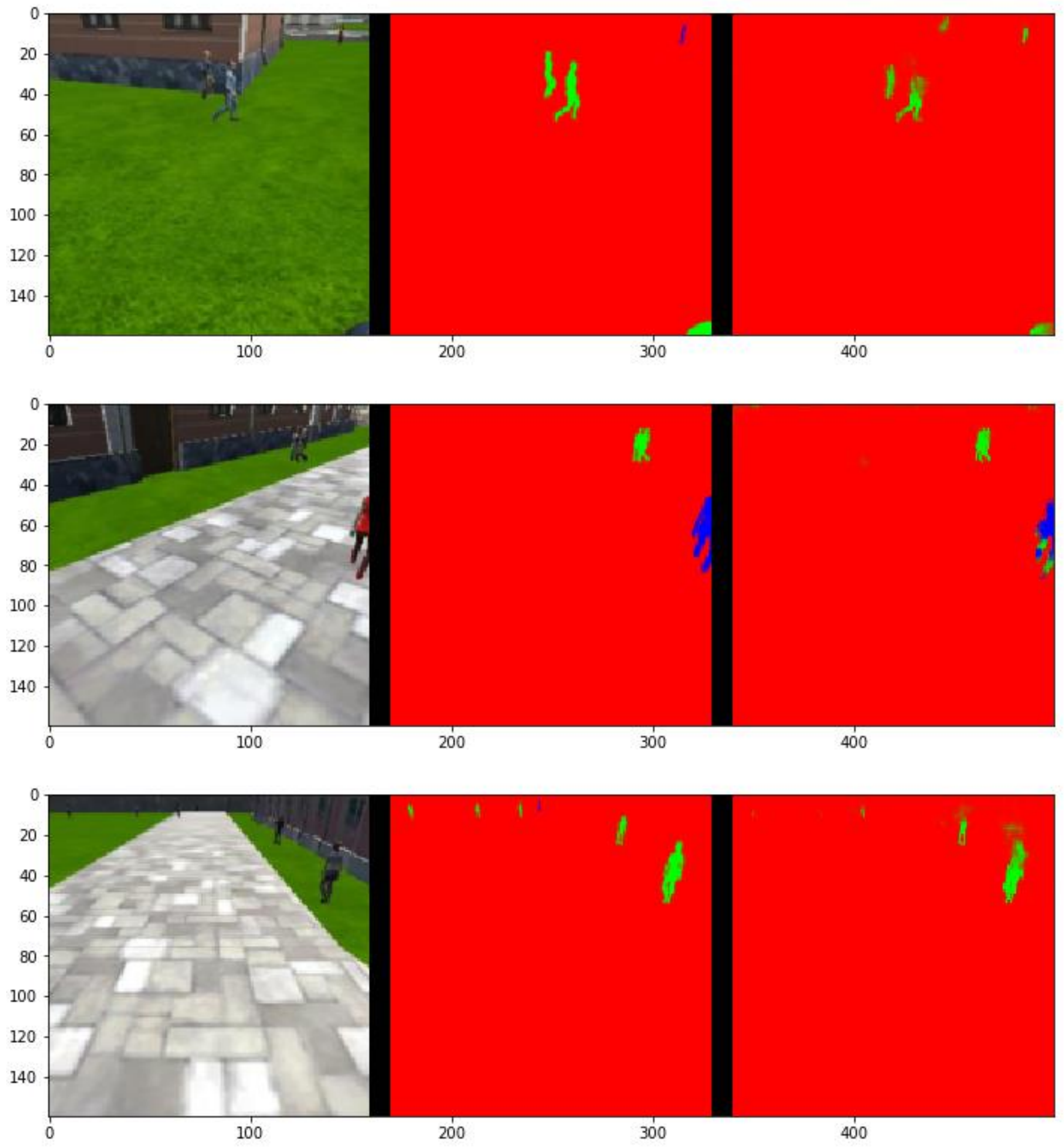
From this training curve, which showed a rapid decline to 0.05 loss over approximately 30 epochs followed by a much steadier decline thereafter, I was able to achieve a final score of 39%. My excitement almost immeasurable at this point, I kept going in training this model. After approximately 300 more epochs, I achieved the following results:



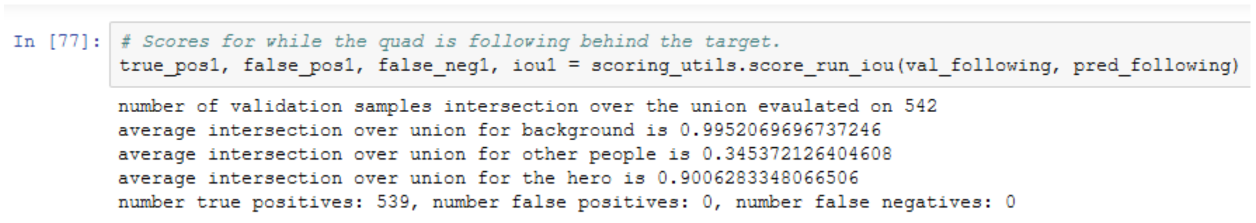

0
20
40
60
80
100
120
140
0
100
200
300
400


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


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

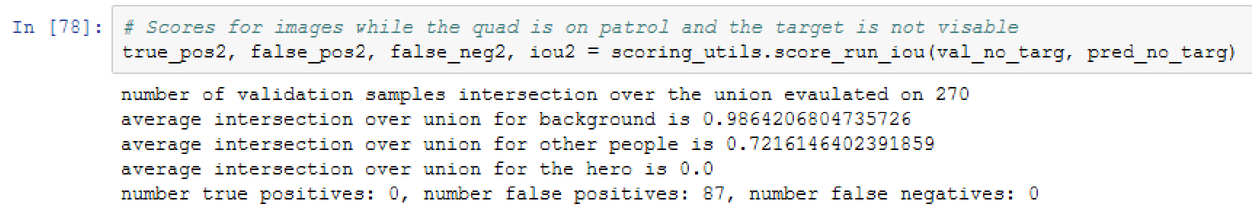


With the drone directly following the target, it can be seen that the neural network trained itself very well. With proper lighting, there is no mistake where the target is, and even with the target in the shade, the model was able to distinguish enough features to properly identify the target. This was validated in the numbers, with no false positive nor false negatives, and virtually all true positives:



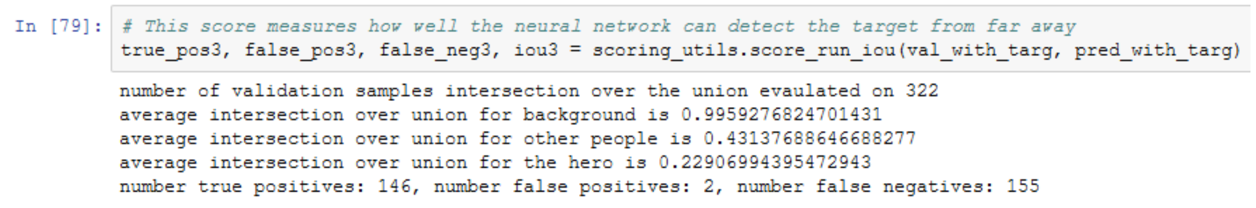
```
In [77]: # Scores for while the quad is following behind the target.
         true_pos1, false_pos1, false_neg1, iou1 = scoring_utils.score_run_iou(val_following, pred_following)

         number of validation samples intersection over the union evaulated on 542
         average intersection over union for background is 0.9952069696737246
         average intersection over union for other people is 0.345372126404608
         average intersection over union for the hero is 0.9006283348066506
         number true positives: 539, number false positives: 0, number false negatives: 0
```

With the test images not containing the target, the model did still have some persistent issues in filtering out features similar to those of the target. Pants of similar color or shape, similar hair, etc. would elicit positive responses from the neural network, showing a few occasional pixels of blue in non-targets. This is illustrated in the data by the remaining number of false positives:



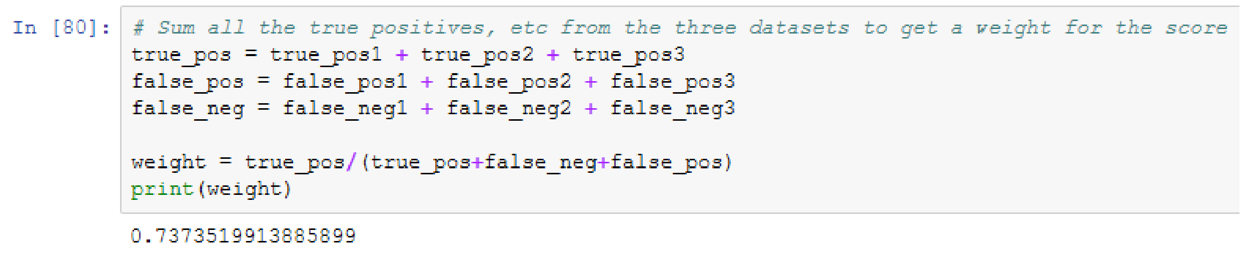
```
In [78]: # Scores for images while the quad is on patrol and the target is not visable
         true_pos2, false_pos2, false_neg2, iou2 = scoring_utils.score_run_iou(val_no_targ, pred_no_targ)

         number of validation samples intersection over the union evaulated on 270
         average intersection over union for background is 0.9864206804735726
         average intersection over union for other people is 0.7216146402391859
         average intersection over union for the hero is 0.0
         number true positives: 0, number false positives: 87, number false negatives: 0
```

With the test images containing the target at a distance, it can be said that the model performed halfway well. Since the drone image processing doesn't account for depth, figures far away from the drone represent fairly few pixels. This makes it far more difficult for the model to properly identify correct colors and patterns. As such, test images that have the target figure close enough to the camera could easily produce positive notation, but having the target very far away was almost guaranteed to be missed by the neural network:

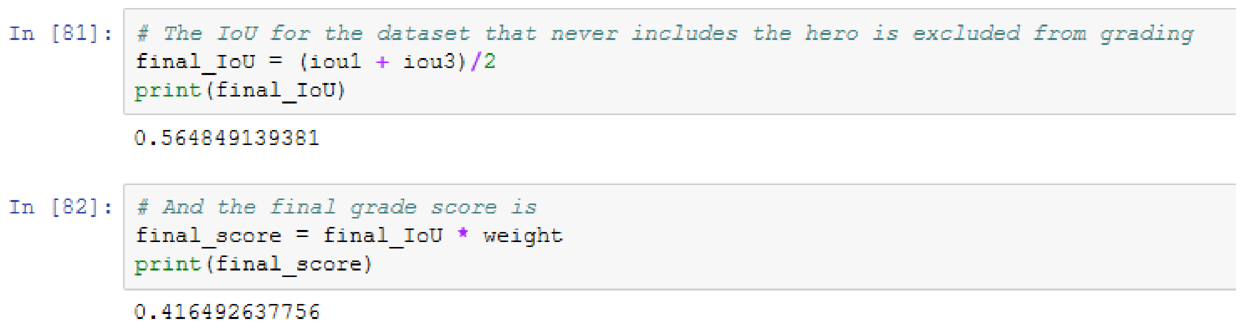


```
In [79]: # This score measures how well the neural network can detect the target from far away
         true_pos3, false_pos3, false_neg3, iou3 = scoring_utils.score_run_iou(val_with_targ, pred_with_targ)

         number of validation samples intersection over the union evaulated on 322
         average intersection over union for background is 0.9959276824701431
         average intersection over union for other people is 0.43137688646688277
         average intersection over union for the hero is 0.22906994395472943
         number true positives: 146, number false positives: 2, number false negatives: 155
```

With this run, the resulting final scores were as such:



```
In [80]: # 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)

         0.7373519913885899
```



```
In [81]: # The IoU for the dataset that never includes the hero is excluded from grading
         final_IoU = (iou1 + iou3)/2
         print(final_IoU)

         0.564849139381

In [82]: # And the final grade score is
         final_score = final_IoU * weight
         print(final_score)

         0.416492637756
```

With a final score of 41.65%, my journey through training my first neural network was finally done!

<u>Fully-Connected Layer vs. 1x1 Convolution:</u>

One of the bigger differences in making the types of neural networks discussed in the course was how the encoder block ended. To start, multiple layers of a neural network were finished with a fully-connected layer. What this did is take the final output of the previous convolutions and transformed the data into a vector of probabilities corresponding to possible objects to identify. The actual number of potential objects, or classes, would define the size of the fully connected layer, but the functionality would remain the same – provide the neural network's best interpretation of the input data. However, one immediate drawback that can be seen is that no spatial resolution is maintained. The fully connected layer may give a high probability that a dog is located in a picture, but it's location within that image will be relatively unknown, since that information would have been lost in making the fully connected layer.

Therefore, a 1x1 convolution can come in handy to allow a neural network to try and derive spatial information as well. In essence, the 1x1 convolution would take the feature map of high level features derived from the encoder block(s) and add an element of non-linearity to it. This would increase the robustness of the existing feature map and allow further convolutions to be run. In this instance, though, with high level features already being the input, these convolutions can expand the map size and connect it to previous layers to give the high level features more spatial clarity. So, the 1x1 convolution acts like a bridge to further discovery and analysis, whereas the fully-connected layer serves as an endpoint (albeit a less complex one). Additionally, the 1x1 convolution can be a rather easy way to alter the depth of the feature map. While the course illustrated the concept that the 1x1 convolution should be kept the same depth as the last encoder output, with decreases in depth following it, I realized in training my most successful model that this 1x1 convolution can function as a bridge to increase the depth of resulting layers as well.

Neural Networks – Pros and Cons:

Neural networks are quite an interesting topic, and their structural similarity to biologic processes is incredibly fascinating to me. However, it is still important to critically analyze this method when considering computational means of, for example, image interpretation.

Given the length of time it took me to train this one neural network, it can clearly be a time consuming process. Not only that, but training a neural network can be a very computationally intensive task. Neural networks require incredibly large amounts of data. For image interpretation, this can require thousands upon thousands of images to train with. For certain tasks, perhaps this is simply not possible. In other cases, perhaps a powerful enough computer is simply not available to train the network in a timely manner. As compared to the previous project, where object recognition was performed via pointcloud data of color and normal vector histograms, neural networks took significantly longer to train.

Of course, the overwhelming benefit of neural networks is the ability to recognize desired objects or images in novel situations. By looking for specific features (versus trying to match exact histogram values), a neural network (if well-trained, of course) can better get at the "essence" of an object. For the previous project's method of object recognition, exact examples were needed (such as a list of images of a soap bar or a box of biscuits); afterwards, the program could match exact copies correctly, but even slight discrepancies in an object of question could lead to wildly different guesses. Neural networks, however, can understand the general idea of an object, such as a cat or dog for example. A neural network may encounter an image of a new breed of dog, but it should be able to still classify it as such because it knows what features to look for. So, from this perspective, a well-trained neural network can show a lower margin of variance in error compared to the previously learned pointcloud method of object recognition.

In regards to training this particular neural network for other objects, such as animals or cars, I don't think too many changes would need to be made. The network would still begin by learning basic features (such as lines or curves) and expand to detect higher level features of those new objects. So long as the training sets and mask are sufficiently detailed, the neural networks should have no problem. Just as this past network had some problem differentiating similar looking people, I'm sure it would have difficulty with similar looking dogs, cats, cars, and whatnot.

All in all, the success of training mostly depends on the quality of the image sets used. Regardless of the desired subject matter (cats, dogs, humans), a comprehensive and varied set of training images will allow a neural network to learn the necessary features of identification.

Future Enhancements:

In terms of neural networks in general, I can certainly think of many ways in which I could try and improve the model used for the follow-me project. However, specifically in regards to the architecture of this project, I'm not entirely sure what more I could logically do to further improve my final scores. Having had spent over a month trying virtually any and all permutations of model structure and parameter settings, I believe I finally managed to find one that allowed me to achieve a passing final score. All other attempts ended up falling short, so all I could really do to improve this particular model would be to continue training it. With a few hundred more epochs, perhaps I could raise the final score to 42% or 43%, but likely not much higher.

Beyond this, given sufficient computational power, the neural network could certainly benefit from using larger-sized images. With higher resolution images, the run-time would certainly be longer, but I presume the neural network would be able to extract more higher-level features. Additionally, having a larger image set in general should help as well. Providing the neural network with more views of both the target and non-targets, I would expect superior results. So, having 10 or 20 thousand images would, presumably, result in superior scores to the 4,000 training images used for this particular project.

With higher resolution images, and more of them, I could also then expand my model. Even with a filter stride of 2, I could maintain enough spatial resolution to set up five or more encoder-decoder layers in the training model. This would allow for extraction of more detailed features and much higher expected scores. I'm not entirely sure how powerful (and how numerous) the GPUs for performing this would need to be, but at least in theory I could perform what is described above.