Follow me project writeup

Introduction

I found this project particularly interesting and challenging. For me the most interesting part of the project was to see how different neural network architectures effect the overall performance. That is why during my experimentation I didn't add any additional training data apart from the data that was provided.

In the beginning I experimented on different neural network architectures. Afterwards I replicated the result on a select few architectures. This was done in due of time limitations as a result of long training times (around 6 hours per architecture) which would take a very long time to replicate all the structures. I ran the training on my computers gpu.

In the first step, when I was playing around with different architectures, I wasn't consistent with the training parameters and overall training time. This is due to the fact that it took me some time to understand which parameters worked. I am still new to deep learning. I also didn't train all the models to their best performance. In the first step I wanted to try different ideas I had and want to get a feel of what worked and provided significant improvements and what didn't. I wanted to have as much iterations as my limited time would allow.

In the second step, when I recreated my results for a few selected architectures, I kept the training parameters the same for all the architectures. This was in order to test performance of the structure as opposed to the training regime.

A few of my neural network parameters were chosen as a result of my gpu's limited resources. I wanted to keep the batch size and steps_per_epoch constant since I didn't fully understand how underlayer training libraries and how these parameters may affect my overall training. Since my gpu was a bit limited this meant that it didn't have enough resources if a certain layer had to many filters. This is why I kept the number of filters to a maximum of 128.

One interesting architecture I tried was to try to create a color encoder which greatly improved my results. This technique can be used on other image segmentation problems where color help the classification process. Even though was a clear improvement I am not exactly sure if this deepness of the network or I had successful created a color encoder. I will talk more about this when I discuss model 2.

Building blocks

My neural networks where all built using the following building blocks. I will give a description of what each building block does as is required by the project rubric.

1. encoder_block(input_layer, filters, strides, kernel_size=3):

   This is a convolution layer. The purpose of this layer is to learn features with spatial information. This is done by sliding at a window with a certain kernel size. The filter size parameter indicates how many new features we extract to the next layer for each “Pixel” in the new layer. We build a small “fully connected” neural network on this small window. We use weight sharing on all these small windows while we slide on the picture. Weight sharing makes sure to extract meaningful features no matter where they appear in the image. This essentially breaks down our image into many image patches for our network to train on. We also select a step for sliding each image. This shrinks the image size. We can look at the convolution as capturing meaningful spatial information.
   The inputs for this layer are:
   a. input layer - from the previous layer starting with the original image
   b. filter – this is the amount of features that are extracted to each pixel of the new layer
   c. strides – this is the number of strides step we are doing when we slide the window. The default value is 1
   d. kernel_size – I added this to the original code. This allows to select the size of window size. The default value is 3
2. decoder_block(small_ip_layer, large_ip_layer, filters):
   This is very similar to the reverse of the encoder layer. The purpose of this layer is to upscale the image back to its original size while extracting the encoded information we found using the encoder for each pixel in the new image. This is done so we can eventually classify each pixel in the original image. The first step is to upsample the input layer (this is done to twice the original size as recommended in the notes). The next step is to add a skip layer. There is a possibility that some important information is lost in the encoding process. That is why on top of the decode we concatenate a ‘skip’ layer. The skip layer (which we have to input to the function) is as if we are “skipping” all the encoding and decoding steps in between the image of the same size what we are upscaling. We concatenate the skip layer on top of the upscaled image. We then add one convolution layers to the concatenated result. This is in order to extract information for each pixel in the new layer. I opted to add two 1x1 conv layers in order to extract non linear information.
   The inputs for this layer are:
   a. small_ip_layer – input from the previous layer
   b. large_ip_layer – this is the skip layer
   c. filters –this is the amount of features that are extracted to each pixel of the new layer
3. 1x1 conv layer:

This purpose of this layer is to extract advanced, non linear features for each pixel in a layer. This increases the complexity of the features we can extract. This is essentially a “fully connected” neural network for each pixel. Like in the encoder layers, the weight sharing gives allot of pixels for our network to train on no matter where on the image they reside. The 1x1 convolution layer is built using conv2d_batchnorm with the following inputs:

  a. input_layer – the input which is just the previous layer
  b. filters – the number of features we want to extract
  c. kernel_size – set to 1
  d. strides – set to 1

4. output layer:
   This layer does the final classification for each pixel to the appropriate class. This is a 1x1 convolution layer with our 3 classifications classes. A softmax is applied to the results in order to get a proper probability.

Difference between 1x1 convolution and fully connected layer

Note: This is for the project rubric

1x1 convolution layer is made using a sliding window with a kernel size of and 1 stride. A 1x1 convolution layer can be seen as a sort of small, pixel wise “fully connected neural network”. By this I mean the input is the “color” depth of the input pixel, and the output is the number of filters. But the 1x1 convolution layer isn’t really a fully connected neural network layer. The main difference is that the 1x1 convolution layer preserves spatial information as opposed to a fully connected layer.

A 1x1 convolution gets as a input a 3d input (4d tensor) and outputs a 3d output (4d tensor). On the other had a fully connected layer flattens the 3d input into a 1d output (2d tensor). Such a layer would make the network unusable since in the end we want to create classification for each pixel in the input image and by removing the spatial information this will deny us the ability to do so.

Chosen neural network architectures

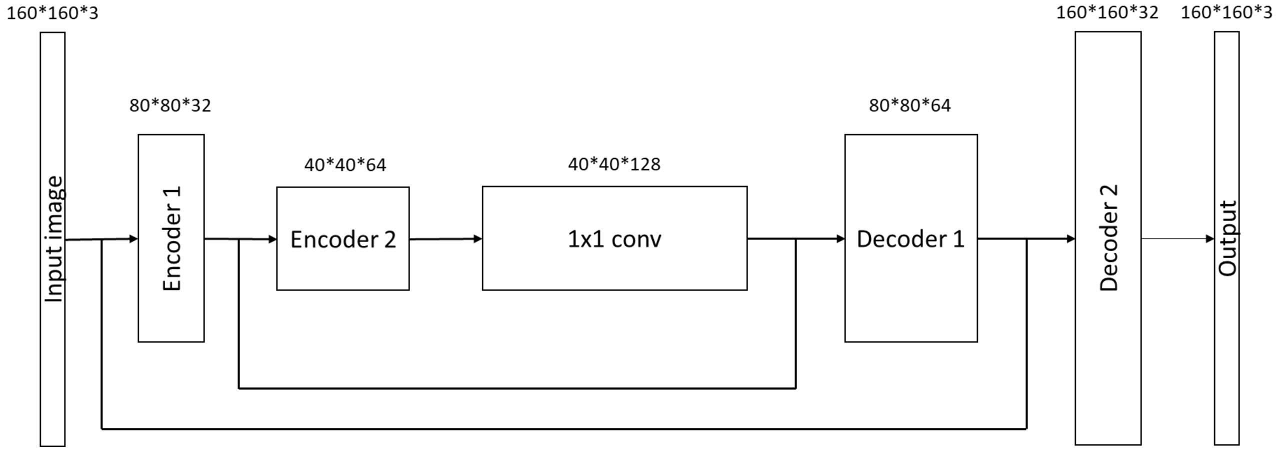
As stated before after experimentation with different architectures. I chose 3 architectures to replicate the results so that I could compare their performance to one another.

Architectures

Model 1

This is the baseline model. This is the model I got at the end of the segmentation lab. The model consists of 2 encoder blocks, one 1x1 convolution layer then 2 decoders

The decoder blocks include skip layers that get from the appropriately sized encoder layers



The parameters for the different layers are as follows:

Encoder layer 1

1. filters = 32
2. kernel size = 3
3. strides = 2
4. input layer size: 160x160x3
5. output layer size: 80x80x32

Encoder layer 2

1. filters = 64
2. kernel size = 3
3. strides = 2
4. input layer size: 80x80x32
5. output layer size: 40x40x64

1x1 convolution layer

1. filters = 128
2. kernel size = 1

3. strides = 1
4. input layer size: 40x40x64
5. output layer size: 40x40x128

Decoder layer 1

1. filters = 64
2. strides = 2
3. input layer size: 40x40x128
4. skip layer size = 80x80x32
5. output layer size: 80x80x64

Decoder layer 2

1. filters = 32
2. strides = 2
3. input layer size: 80x80x64
4. skip layer size: 160x160x3
5. output layer size: 160x160x32

Output layer

1. filters = 3
2. strides = 2
3. input layer size: 160x160x32
4. output layer size: 160x160x3

I chose these values for this neural network, especially the number of filters, randomly and got good initial results.

Further on during my experimentation I built a neural network with smaller filters (half the number of filters for each layer) with the same training regime that I used to train this network. I got worse results it seems that I was fortunate in picking these initial values.

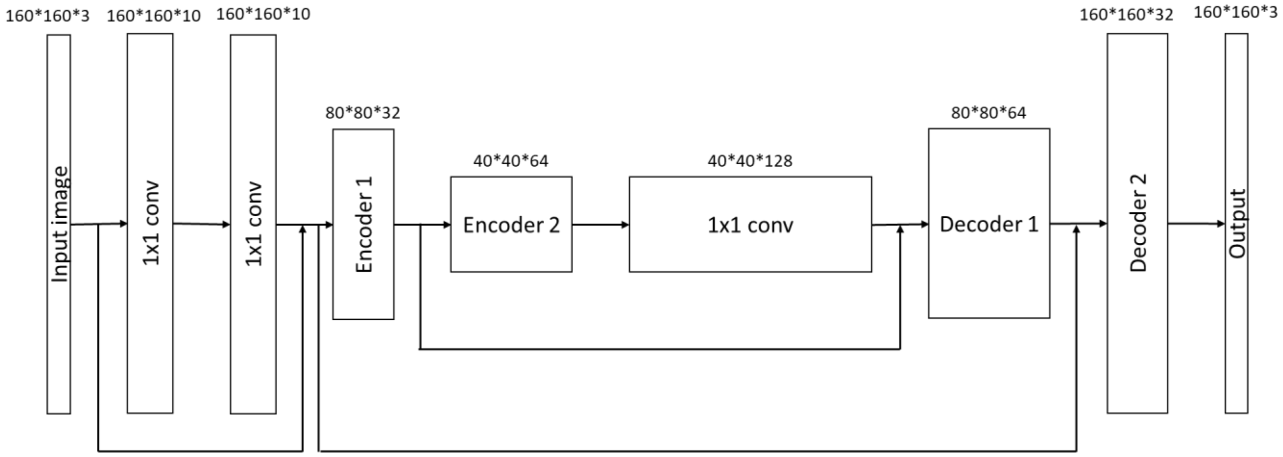
During my experimentation of different neural networks I also tried networks with more features. My gpu limited me with networks with layers than 128 (256 for example). It gave me an error of resources. That meant that I had to change my training parameters, especially the batch size and the steps per epoch. I wanted my training to be the same (and fast enough) for all my networks. That is why I decided to limit the number of filters I use for a given layer.

Model 2

This model is the same as the model 1 but with two extra 1x1 convolution layer right after the input image. The original image is then concatenated on top of the results. This is like adding a 'skip' step. The idea here is that I noticed that in the color in the original image was very important and could assist with classification. The hero is mostly red, the background is gray, green and black, etc. The idea was to have the neural network learn extra meaningful color channels. It would then remap the pixels to this new color space which will help it in consecutive layers. During my experimentation I tried only one 1x1 layer but didn't see significant improvements. That is why I used two 1x1 convolution layers in order to get higher non linearity. I added the input 'skip' layer since could still hold some important information.

The parameters for this neural network are the same as model 1. The only difference is the number of color channels of the 1x1 convolution layers. I added 10 color channels (filters). During my experimentation I tried neural networks with 5 and 20 color channels. I received the best results with 10.

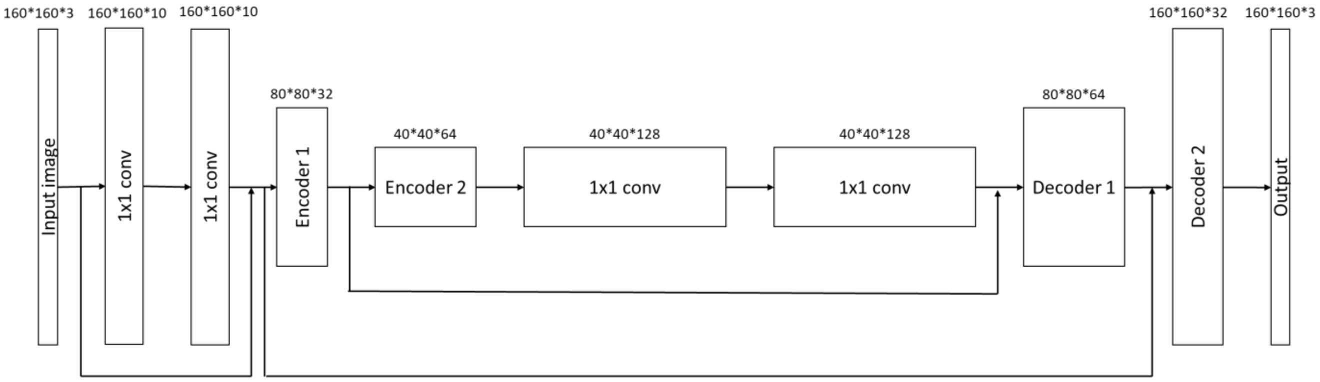
Another difference is that I used the output of the 1x1 color encoder layers as the skip layer for the final decoder block.



Model 3

Next I tried to make the network deeper. I took model 2 and added an extra 1x1 convolution layer in the middle of the network, between the encoding and decoding layers. This was done in order to make the network deeper and to try to add a higher degree of non linearity where the network is extracting the most information.

The parameters for this model are the same as model 2



Training

I used the same training regime and parameters for all the models. I did this because I wanted to be able to compare the performance of the network architectures.

Each model I trained: 20 epochs at a learning rate of 0.001 and then another 10 epochs at 0.0001. I came to these results after allot of experimentation. Running the second training at a lower training rate greatly improved the results. For some reason while running my training I never experienced overfitting. I didn't see the validation loss go up to much. I guess this was because my networks where complex and I didn't run enough epochs in order to experience overfitting.

The other values I used where:

1. batch_size = 20
2. steps_per_epoch = 250
3. validation_steps = 60
4. workers = 2

This batch size was chosen because of the limitations of my gpu. I wanted my batch size to be high enough so the training would be fast enough. I wanted the maximum number features I could have in my network and the steps per epoch to be reasonable. I tried different batch sizes and eventually settled on this batch size. The steps per epoch and validation steps where chosen based on calculations. The total number of training images 4131 and validation images 1184. The steps_per_epoch was chosen so that:

batch_size*steps_per_epoch > 4131

For validation steps:

batch_size*validation_steps > 1184

The number of works was chosen as 2 because it was the maximum my gpu could handle.

Results

I will start off with some numbers:

The overall scores

Model	Loss	Validation loss	True positive score	Average IOU	**Final score**
Model 1	0.0252	0.0396	0.731	0.524	**0.383**
Model 2	0.0195	0.0253	0.768	0.58	**0.446**
Model 3	0.0191	0.0267	0.776	0.579	**0.449**

Some stats for pictures while the quad is following behind the target

Model	IOU background	IOU other people	IOU hero	True positives	False positives	False negatives
Model 1	0.994	0.295	0.867	539	0	0
Model 2	0.995	0.338	0.891	539	0	0
Model 3	0.995	0.342	0.894	539	0	0

Picture while the quad is on patrol and the target is not visible

Model	IOU background	IOU other people	IOU hero	True positives	False positives	False negatives
Model 1	0.982	0.626	0.0	0	66	0
Model 2	0.985	0.686	0.0	0	64	0
Model 3	0.985	0.708	0.0	0	57	0

Some stats for pictures while the target is far away

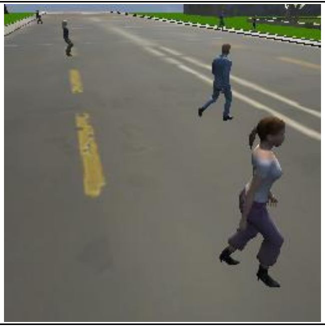
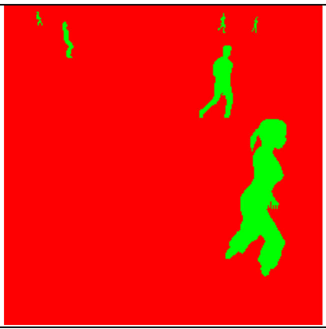
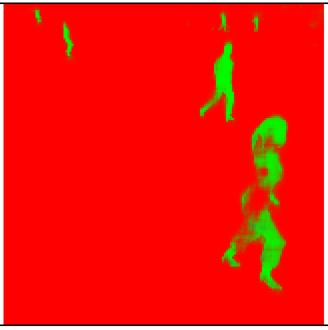
Model	IOU background	IOU other people	IOU hero	True positives	False positives	False negatives
Model 1	0.996	0.393	0.179	124	1	177
Model 2	0.996	0.441	0.269	158	3	143
Model 3	0.996	0.446	0.264	158	1	143

Analysis the results

The biggest improvement for the final score was in between model 1 and model 2. There was only a very slight improvement between model 2 and 3. It can be noted that this is a significant improvement of the score. It is unclear to me if the improvement is because I made the network deeper or because of where I placed these layers. In other words if adding two extra 1x1 conv layers elsewhere would improve the results as much or if it was because of the location of those two layers really created meaningful color encoders which helped the classification (as my theory). It would be interesting to run a network with two 1x1 conv layers placed elsewhere in the network and to see if there are comparable increased results.

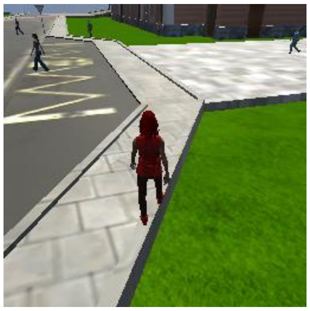
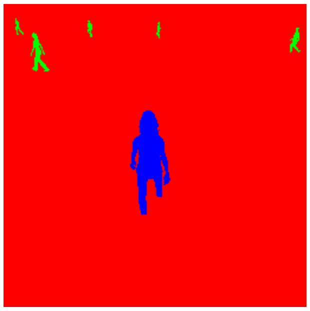
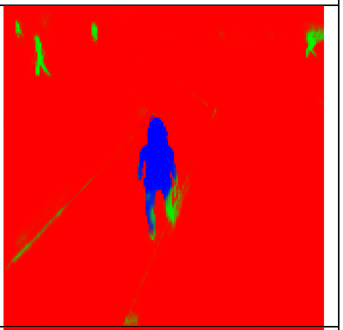
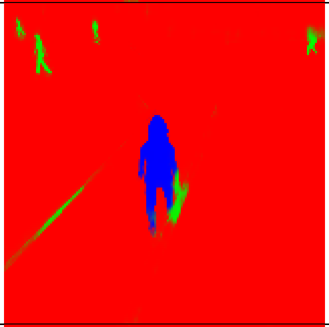
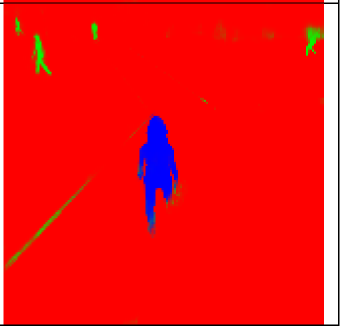
When analysing at the results a few interesting things can be seen.

In some images where the people are close (and especially close to the edge of the screen) there are “holes” where the neural network recognizes the middle of the body as background like the following.

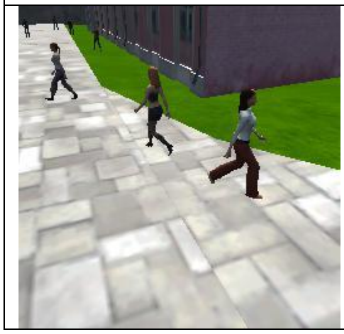
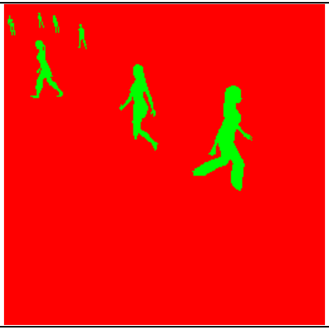
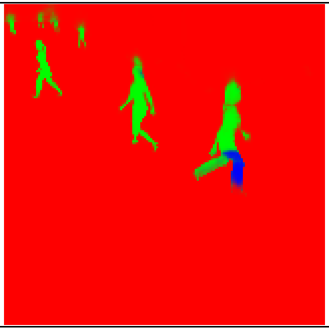
Picture	Mask	Neural network prediction
		

During my experimentation I tried to play around with the kernel size of my encoding layers to see if I could get better results but it didn’t help. The idea is that if the kernel size is larger than the neural network will take into account the surrounding pixels when classifying a certain pixel. Maybe that way it would eliminate these holes. When I experimented I changed the kernel size along with other changes I experimented with, so it is unclear if it really doesn’t increase performance. Also in those early stages I focused on the overall score and didn’t look to much at the images, so it might be interesting to revisit kernel size.

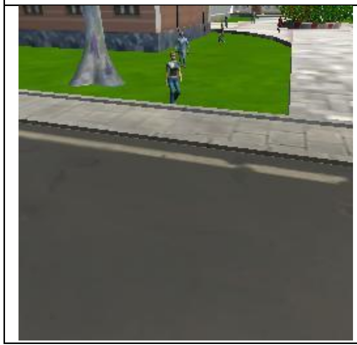
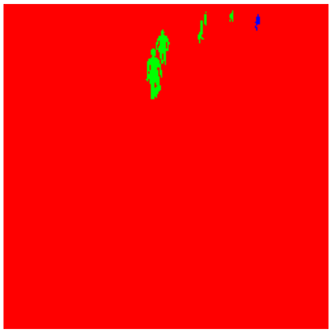
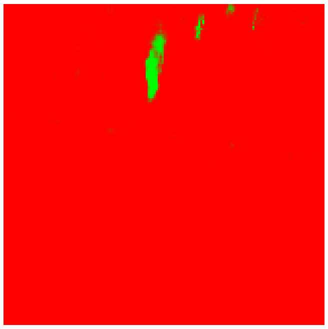
Another interesting thing I noticed was the neural network was using color as an aid to classify pixels. This was evident in all the models. It is evident in misclassification. As you can see in the following images the edge of the sidewalk is classified as “other people”. This can be seen in other sidewalk images. After looking at a few of such pictures it seems that the neural network looks for something black at a certain angle. It can be seen in other misclassifications that small dark objects (like the hero far away) tend to be classified as “other people”.

Model	Picture	Mask	Prediction
Model 1			
Model 2			
Model 3			

Also the neural networks have a higher tendency to misclassify brown objects as hero (like pants of certain other people). This image is for model 3.

Picture	Mask	Neural network prediction
		

When we look at the score and the way it is calculated and what hurts the overall score the most is average IOU especially the IOU when the hero is far away. I changed the original code so that I could see the images which were tagged as false negatives to see where the neural network fails to work. When I looked at those images I saw that in almost all the images either the hero was occluded (either by coming off the edge of the screen or by other people) or the hero was very far away. In any case the hero most almost always misclassified as “other people”. I found this surprising since in some images I find it hard to see the hero. Any person far away look similar, and have a tendency to be dark and small. The networks must have a bias of classify anything small and dark as “other person.” This may be due to the fact that the training data contained a small amount data where the hero was far away as opposed to other people far away which we have many examples of in the training set. If I added samples of the hero far away to the training set it would probably improve the results.

Picture	Mask	Neural network prediction
		

Limitations of the neural networks

These neural networks I trained are very biased to the training images I used. That is why it behaves badly when the hero is far away and misclassifies the hero as 'other person', since there is a lack of examples. If I wanted the neural network to follow another object (dog, cat, car, etc.) instead of a human I would have to provide the model with appropriate training data. That means I would have to record images (enough images) of objects I wanted to track.

Even if were to change the scene where the hero was walking (like a in a forest or a desert) or were to introduce apart from the "other people" things such as animals, this could greatly impact the results of my trained network. I would have to retrain the network with examples of the environment I want the network to work on.

Future enhancements

There are a few things that can improve the results. The most significant is adding more training images. Especially if I use more images where the hero is far away. I could also flip images to create more examples. This would probably greatly improve the results of the network.

Another improvement that could help is change the number of filters (color channels) for the first two 1x1 conv layers. Apart 10 channels that I eventually used in models 2 and 3, during my experimentation I tried 20 and 5, the results weren't better but other values could yield better results.

Playing around with the kernel size of the encoders could yield better results. This could maybe solve the 'holes' in classifications I talked about in the analysis.

During my experimentation I tried to lower the number of filters by half. The results I got were worse. Maybe increasing the size of the filters could improve the performance. Maybe making the network deeper yield much better results. I doubt this would work since when neural networks are very deep back propagation has difficulties making changes to the first layers.

Training the model for more epochs might improve the performance but probably not much.

Very important Note!!!

1. **The md5 file in the weights folder are the weights for model 3. The weights for the other models are in subfolders**
2. **In order to get the code work the cell that contains the fcn_model function for "model 3" cell has to be run and not any of the other cells**