Deep Learning - Follow Me Project

Introduction

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

Data Collection

As suggested in the lesson notes I took some extra data inside the simulator. The first run has 2 patrol points for the drone, with a zigzagging hero path underneath, as shown in Figure 1. The second run has the drone following the target navigating a complex path, with many distracting crowd figures also present, as shown in Figure 2.

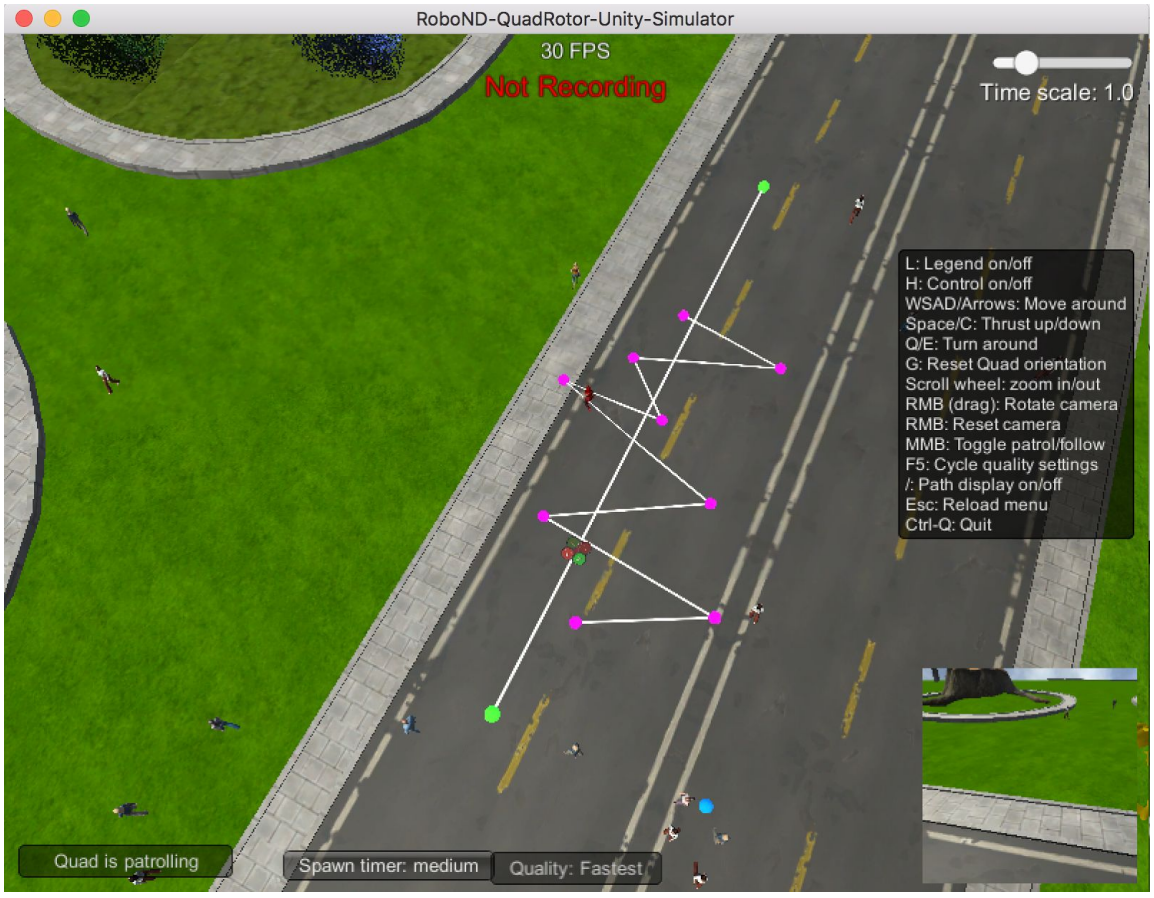

Figure 1 - Data collection run with the hero zig-zagging underneath the patrolling drone

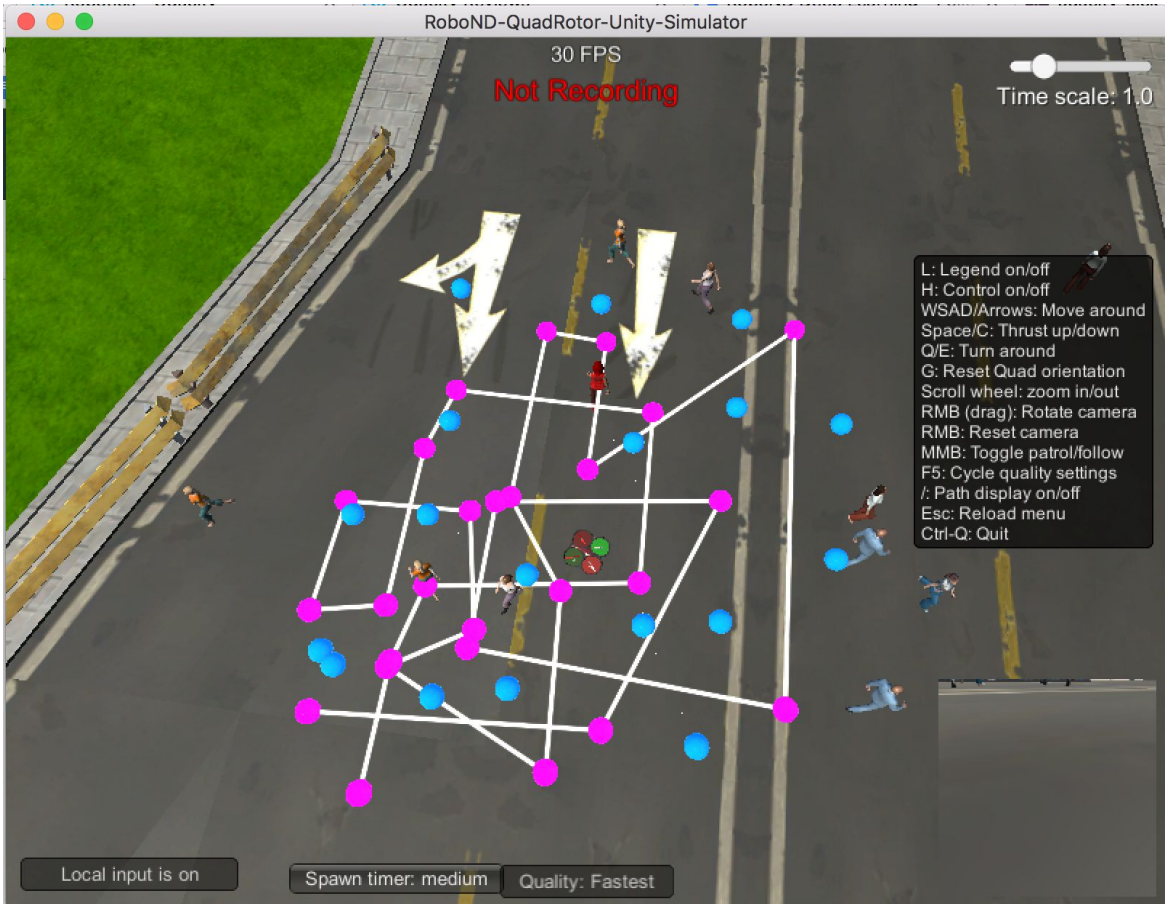

Figure 2 - Data collection run with drone following the target with complicated path, with many crowds

Network Architecture

Until now I have mostly used neural networks that include at at least some point a fully connected layer(s). This kind of architecture is useful for tackling tasks such as image classification, which just want to determine ‘what’ is in a certain image. However in this project we want to know ‘where’ in the image a certain object is, and so fully-connected layers will not work since they remove any sense of spacial information. Thus we need to use a fully-convolutional network, where every single layer in the network is a convolutional layer.

The fully-convolutional neural network I used in this project is comprised of two main parts, an encoding stage and a decoding stage, which are connected via a 1x1 convolutional layer. The encoding stage exists to extract features that will be useful for segmentation from the image. It

does this via multiple layers that start by finding simple patterns in the first layer and can gradually learn to understand more and more complex shapes and structures in the deeper layers. The 1x1 convolution layer then seeks to implement the similar function as a fully-connected layer would, except with the advantage that it maintains the spatial information. It connects to the decoder stage, which again consists of multiple layers which up-scale the encoder output back to the same dimensions as the input image. Along the way there are also skip connections, which connect some non-adjacent layers together. For example the output from the first encoder is connected directly to the input of the final decoder. This is done to retain some information that would be otherwise lost during the encoding process and hence can allow greater accuracy and precision of the final segmentation. Finally after the last decoder stage there is a convolutional output layer with softmax activation to make the final pixel-wise segmentation between the three classes.

The final fully-connected neural network architecture I used is shown in Figure 3. It features 2 encoder layers and 2 decoder layers. Also, inside the decoder there are 2 separable convolution layers after the upsampling and concatenation steps (not shown in figure). This was found to be the optimal network architecture after running several experiments, which I detail later in the Experiments and Results section.

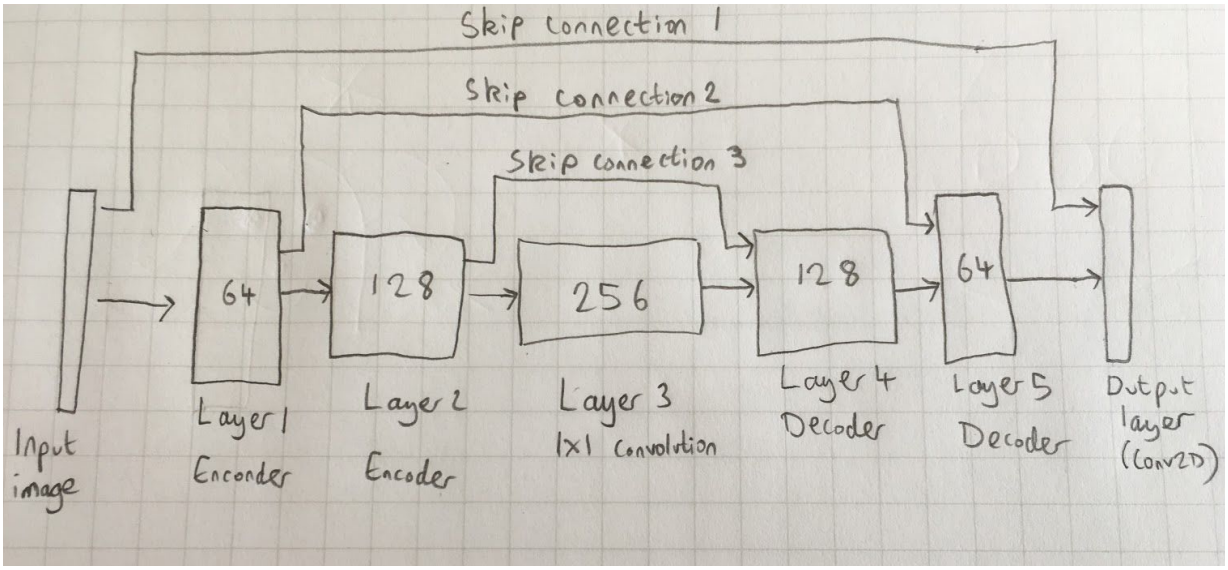

Figure 3 - Final network architecture

Training Process

Fully convolutional neural-networks are very computationally intensive to train and I know they don't run efficiently on CPUs. Nevertheless as a benchmark of performance I decided first to

have a try training on my CPU (2.5GHz i7, 16GB RAM) to see just how badly it performs. I attempted to run for 10 epochs, 500 steps per epoch, with batch size 64. After 23 hours (!) maxing out the CPU it had completed 9.5 epochs successfully. Clearly this is not efficient, but at least it was working and training successfully, as seen by reductions in both the loss and val_loss scores. After nearly a day of struggle I was excited to see the CPU to complete the task. But then disaster struck and my computer crashed! Obviously running flat out all day wasn't a good idea and it got so close, but so far to completing. I've heard many times that CPUs are not good for this kind of task, but it was good to experience it first hand once, to fully appreciate it.

To counter this we must train on GPUs instead. The best option would be to use AWS, however unfortunately they haven't responded to my limit increase for the p2.xlarge instance yet, so this was not an option. Very graciously one of the class mentors (@shreeyak on Slack) owns a good GPU and allowed me to ssh into his system so I could run my notebook on his computer. I pushed my notebook to Github, added him as contributor and then cloned it again onto his PC. Then through the magic of SSH and port forwarding I could get the notebook to run on his GPU whilst appearing on my browser - very cool!

Immediately I could see the difference in power. Epochs that used to take 1-2 hour to complete on my CPU took around 5mins or so on his. The whole training process completed in about 1-1.5 hours, rather than the day it took me. So in total there was around x20-30 speedup just by switching to GPUs!

Finally once I completed the training I pushed the model weights and notebook back to Github, from his machine on his account. This is why there are some extra commits in my repo that appear not to be from me. It is all my work, just coming from his account since that's where it was running. Hope that makes sense.

Experiments and Results

I first decided to run a simple network, with just 2 encoder and 2 decoder layers, and one separable convolutional layer inside the decoder. At first it is better to keep simple, for many reasons: to make sure the overall network implementation was done correctly and can train, also partly due to the long training time and also to avoid potential overfitting. It is always possible to increase the depth and complexity of the network later if it runs ok. The batch size was set to 32. I would prefer to use a larger value, but any more than this then the GPU ran out of memory so it didn't work. I set learning rate to 0.01 - in general a lower learning rate could be better, but the lesson notes mentioned that since we implement batch normalisation a higher learning rate can be acceptable. So this is a starting point which can be optimised later. Steps per epoch was left as default 500 and I ran for 10 epochs. I think it will be interesting to see how the loss and val loss score changes throughout the run and hence determine whether more or

fewer epochs would be better for later runs. Although more than 10 epochs would be a little difficult given the already long training times.

The final IOU score of this first attempt was 0.36. Not too bad, but could do better. I decided to test the model (model1_weights.h5) in the simulator to see what happened. The drone was able to lock-on to the target and could follow for a little while, but whenever the hero made 90 degree turns the tracking broke down and could no longer keep up. For a first attempt it was promising that the basic functionality could work, but I know it needed to do better than this in the end.

For the second test I just made one simple change, reducing the learning rate from 0.01 to 0.005. This could mean the network takes longer to train, but equally that it could eventually arrive at a better overall score. It did indeed make an improvement, producing a final IOU of 0.38, so around 2% increase in accuracy. The exported notebook of this run can be seen in the supporting file a1_model_training.html and the weights in a_model1_weights.h5, but I decided to make further improvements before testing again in the simulator.

For the third test I kept learning rate at 0.005 since it made an improvement in the previous run. The other main change I made was to add an additional separable convolutional layer to the output of the decoder stage (i.e. increase from one to two). Also, in order to get an update the graph and check on the val_loss score more often, I reduced the steps per epoch to 100 and increased the number of epochs to 100. This result in about double the amount of data being processed, and also increase the graph update speed by x5. Finally I replaced the call to the default plotting callback in the notebook to use Tensorboard instead to get a clearly overview of how the network is training. This was done by modifying these lines of code:

```
#logger_cb = plotting_tools.LoggerPlotter()
logger_cb = keras.callbacks.TensorBoard(log_dir='./logs', histogram_freq=0, batch_size=32, write_graph=True, write_grads=False, write_images=False, embeddings_freq=0, embeddings_layer_names=None, embeddings_metadata=None)
```

The Tensorboard graphs of loss and val_loss can be seen in Figure 4. Overall these changes lead to another improvement in the network, reaching a final IOU score of 0.41, a further 3% increase. I took the weights and config generated by this network (a_model2_weights.h5) and retested in the simulator. This time it worked really well! The drone locked on to the hero and consistently tracked her without any difficulties, even when she turned sharp corners or disappeared into a busy crowd! I made a short video of the drone in action - you can see it here:

https://www.youtube.com/watch?v=NHvHV9I___s

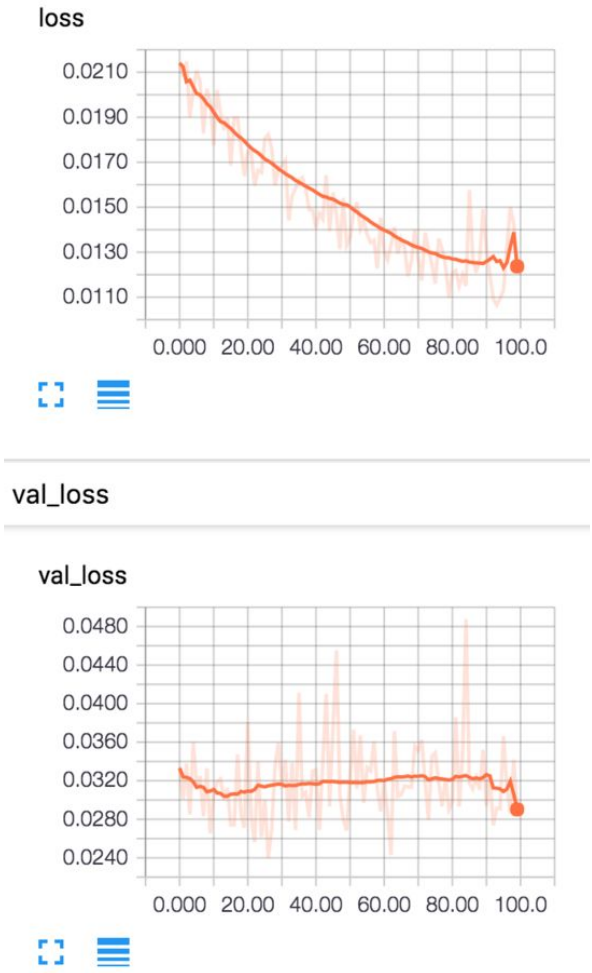

Figure 4 - Tensorboard graphs of loss and val_loss for the third experimental run

Although the simulator was now working correctly and I had completed the project, I decided to do one more run whilst I still had GPU time available to see if I could improve the IOU score any further. I kept everything the same as the previous run, except I added one extra encoder layer and one extra decoder layer, so in total the network became 2 layers deeper. I was interested to see the impact this would have on total training time and also whether the more complicated model would lead to a better final result, or if conversely it would lead to overfitting.

It turned out there was no major change in training time, in all cases it took about 1.5 hours to complete. As for the final result, it got an IOU score of 0.37, but more importantly the val_loss score increased slightly after 50 epochs or so (whilst the loss kept decreasing), which suggests overfitting was taking place. This could potentially be overcome by using even more data, but for now it means my best model was the one from experiment 3 and so this is the final one I present for review (a2_model2_weights.h5).

Conclusion

In conclusion the drone is successfully able to track and follow the hero target in simulation! It is quite satisfying to watch it continuously track the hero even when she makes sharp turns or disappears into a big crowd. That's the power of deep learning! This network architecture could be used to track other objects too, such as dogs and cars, however additional data would be required and the network trained again using those images. Making the drone be able to distinguish between these different types of objects and then be able to switch between them on the fly (no pun intended) would make for an interesting future extension to this work.

Furthermore, even though the simulation works well and achieves the intended result, the final IOU scores seem a little lower than I would like. I think it could be possible to improve these scores with an increase of training data, especially related to other people (aside from the hero) which is what the current model is currently worst at determining (hero and background are both very good). Unfortunately until AWS approves my limit increase request it is not possible for me to do any more training at this time, but I would be interested to try again in the future to improve my scores. Experiencing the difference between CPU and GPU first hand was a really good learning point of this project.

Overall this project was good fun and a great introduction to the application of deep learning in robotics - thanks!