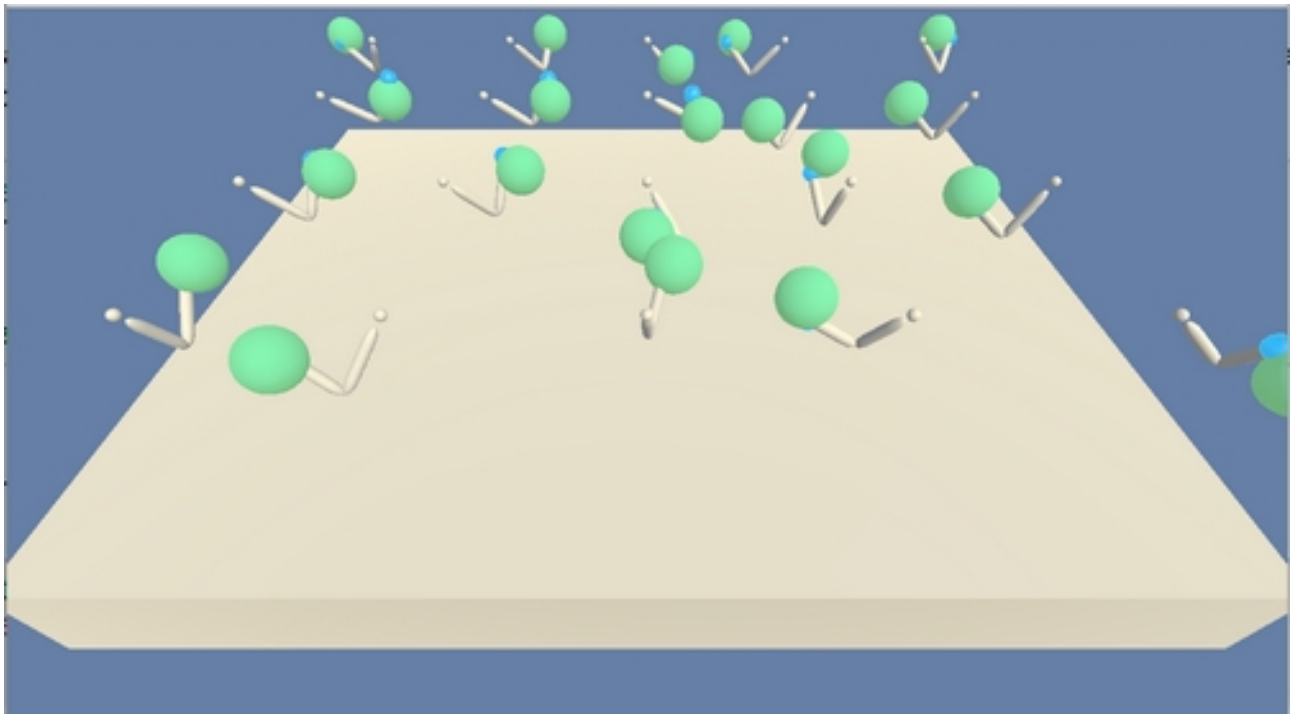
Udacity Deep Reinforcement Learning ND

Project 2 – Continuous Control (Multiple Robot Version)

REPORT



Learning Algorithm

The final agent, was trained using an Advantage Actor Critic Method or A2C.

Hyper Parameters

The following hyper-parameters were used , as explained later.

GAMMA	0.99	Discounted learning rate
LR	0.00025	Learning rate for ADAM
N_STEPS	8	Number of steps to collect
ENTROPY_WEIGHT	0.001	Entropy weight modifier

Network Architecture

The algorithm used a neural network that had the following structure.

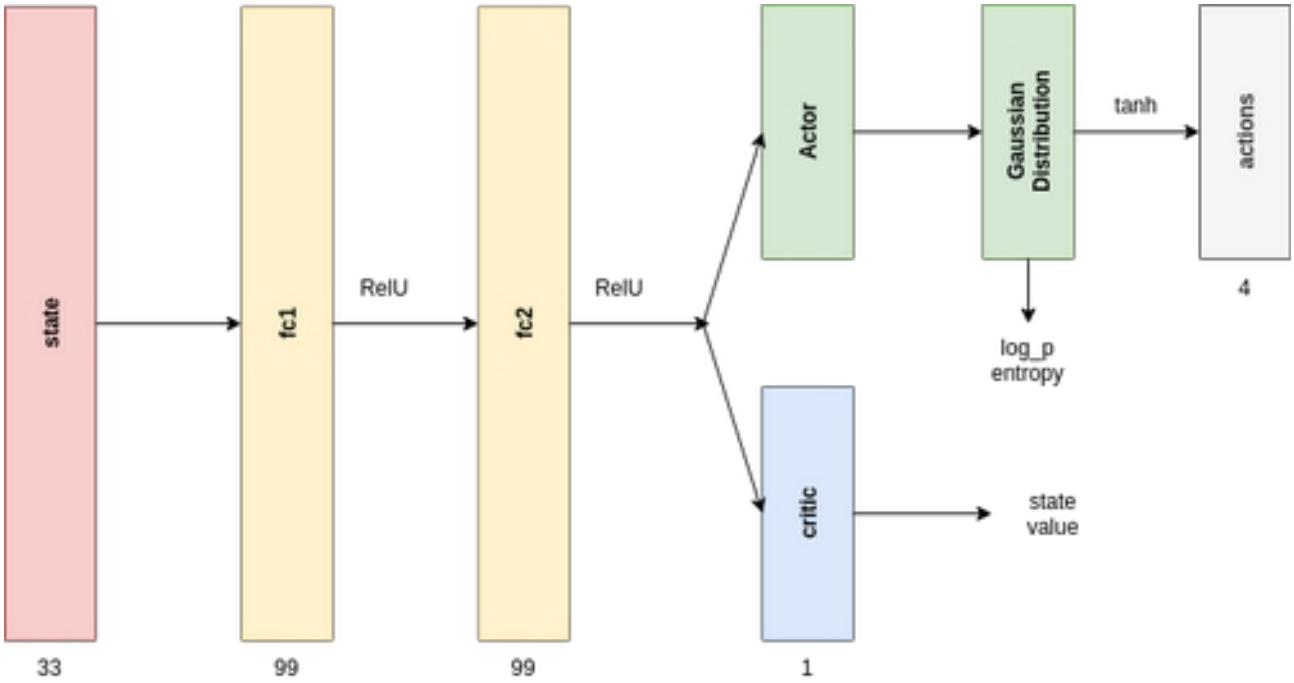


Figure 1: Neural Network Configuration

The input state is passed through two fully connected layers of size 99 and 99 respectively, using rectified linear unit as the output function of each. The final output of this can be passed both to the Actor or Critic layer as needed.

The Actor layer is just a simple Linear Activation layer (with bias) and 4 outputs , this is passed through a trainable normal distribution layer (the standard deviation is adjusted , by the optimizer) ,

The Critic layer also takes the same output from the previously calculated two fully connected passes , and produces a single output value of the state value.

Training Code

The algorithm collects N_STEP (very small) samples from each agent every episode. The pseudocode for training can be seen below ...

```
Initialize network model Ω with random weights.
FOR each episode
     reset environment and get initial state s
     set batch counter b = 0
     WHILE episode still running
          a = Ω.actor( s )
          s' , r = perform action tanh(a) on environment
          batch[b++].{s,a,r} = s,a,r

          IF b >= N_STEPS

               r = Ω.critic( s' )
               WHILE b >= 0
                    r = batch[b].r + GAMMA * r
                    batch.vt[b--] = r
               ENDWHILE

               v = Ω.critic( batch.s )
               log_p,ent = Ω.actor( batch.s) , batch.a )

               td = v – vt

               value_loss = mean( td ^ 2 )
               policy_loss = - mean( log_p * td )
               e_loss = mean( ent ) * ENTROPY_WEIGHT

               total_loss = value_loss – e_loss + policy_loss

               optimise Ω to reduce total_loss

               b = 0

          ENDIF

          s = s'

     ENDWHILE

ENDFOR
```

Plot of Rewards

A plot of the score, for each episode while the agent was training is seen below. Each episode score is the average for all twenty agents in that episode.

The agent was trained, according to the definition at least after 267 episodes.

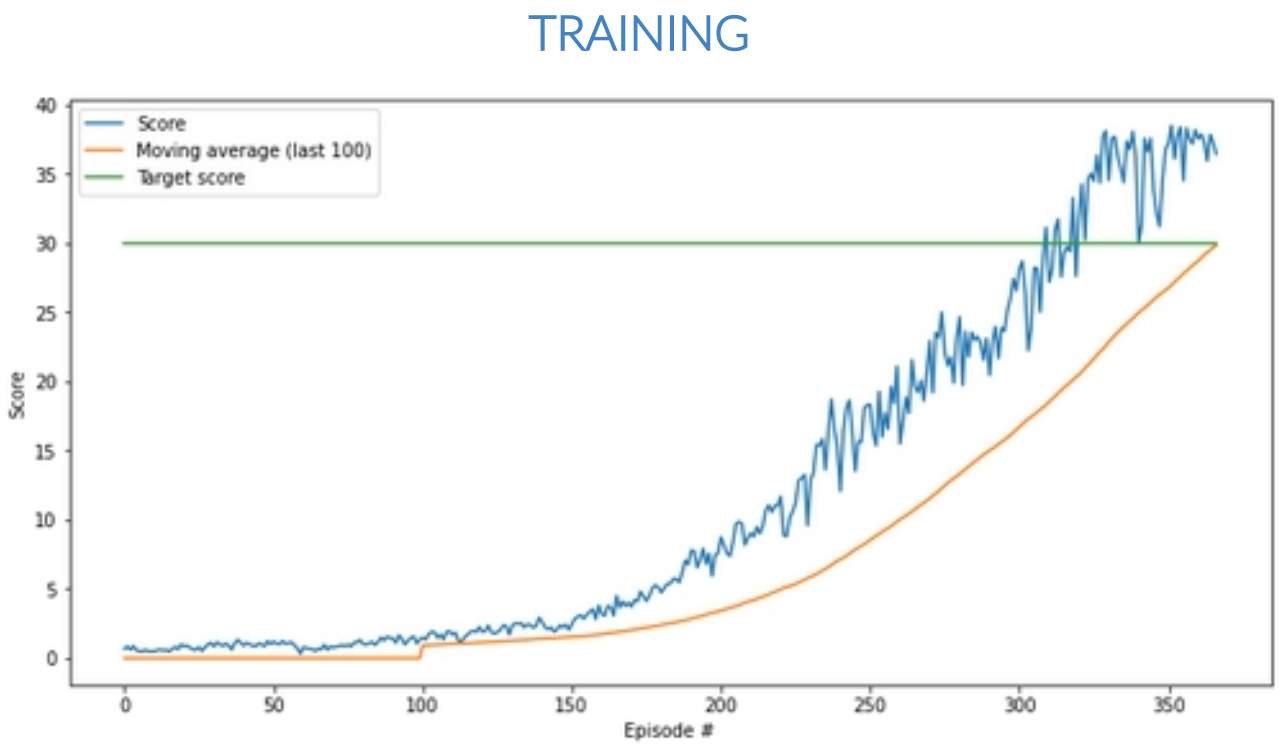
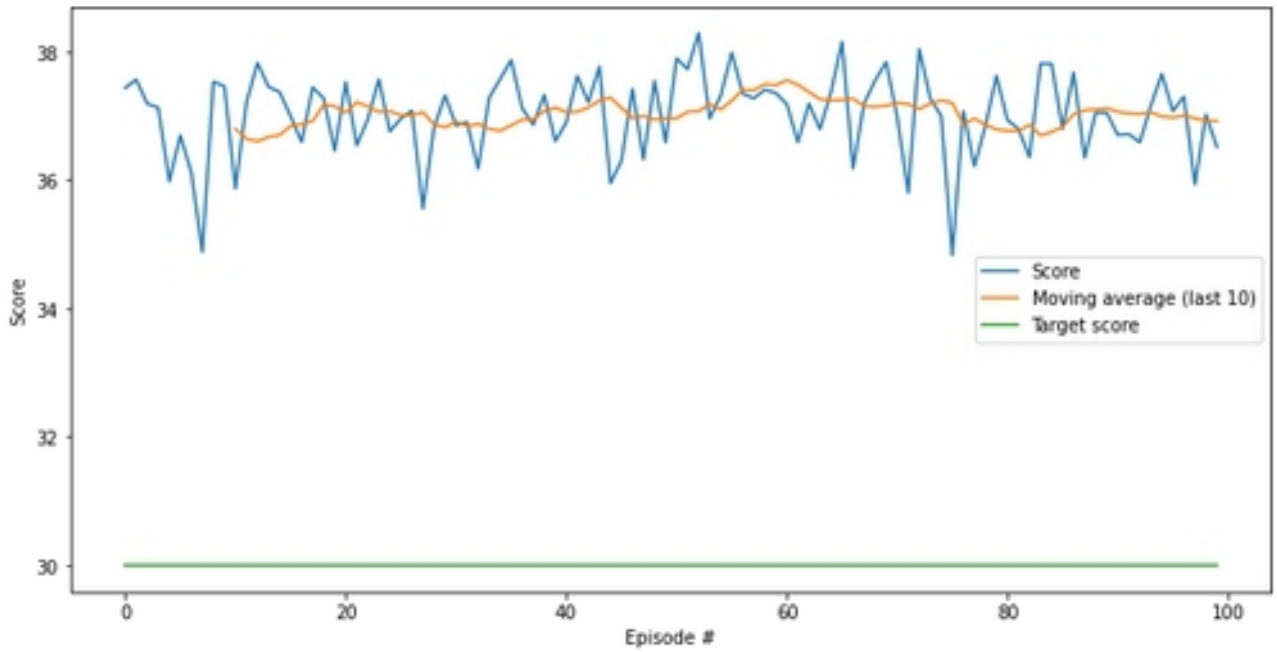


Figure 2: Plot of mean of mean of agent rewards per episode

The orange line , showing the average over 100 episodes , is very smooth – and I think this is due to use of the entropy loss, smoothing out the more violent changes in score.

Running another 100 episodes , after the training has finished can confirm that the model did not degenerate after the training conditions were meet (it can happen) . In this case we seem to be still getting an average of 37 or so.

VALIDATION RUN



Ideas for Future Work

For future work, we can probably improve performance of the agent somewhat. For execution time improvement the code could be optimised somewhat (although the bottleneck does seem to be the speed at which the simulator can provide samples).

The hyper-parameters were roughly tuned, to provide acceptable results (IMHO) , but further tweaking given time could yield better results. The current version does use entropy loss in it's loss function, but it should be fairly simple to add GAE (Generalized Advantage Estimation) to the update routine however time is not unlimited.

Normalising the state input vectors , and may be the reward functions.

My solution uses the ADAM to optimise the solution, with a very small learning rate , other optimisers may perform better.

Future note to self.

Other techniques such as using D4PG , PPO , TRPO and DDPG may yield better / quicker results, and may be worth further experimentation when time permits.