Deep Reinforcement Learning Nanodegree

Project 2 Continuous Control “Reacher”

Ben Hosken - November 2018

Overview

This document provides details of the process, learnings and results from the Continuous Control project completed as part of the Deep RL Nanodegree.

In this project, the aim was to move a double jointed simulated robot arm to target locations. A reward of 0.1 was achieved for each steps that the arm was in the goal position. The observation space consists of 33 variables specifying the position, rotation and velocities of the arm. Each action is a vector of four elements with a continuous range from -1 to 1.

In this project, we used a single agent environment. There is also a 20 agent environment that could be used. The single agent environment is considered solved if an average score of over 30 is achieved for 100 consecutive episodes.

Ultimately, the problem was solved using a DDPG algorithm with a two hidden layers (400,300) with reward clipping, batch normalisation and minor adjustments to the agent parameters. The two improvements based on testing in our environment were not adding noise, and reducing the batch size to 64. Adjustments to the L2 decay, learning rates, increasing model width or depth or replay buffer experience frequency had negligible or even negative impacts on the performance.

Approach

I used the Deep Deterministic Policy Gradients algorithm for this project. DDPG is an actor-critic, model free algorithm that works well for continuous action spaces. The actor component focuses on learning the policy trajectories, while the critic learns an estimated value function.

There are two neural networks within the DDPG algorithm. The actor estimates the policy and the critic estimates the value. Experience replay buffering is used to avoid over correlation in the learning. Soft updates are made to the target networks and replay sampling can be done each episode or after a number of episodes.

I began the project utilising the provided DDPG agent and model implementation code; adjusting the setup and training loop code for the Reacher environment rather than Gym. After seeing no learning I realised I had made a simple mistake of setting the state to the new state prior to the learning step. This school boy error cost many hours of fruitless experimentation.

After seeing training progress successfully I began exploring the hyper parameter space. Based on discussions in the forum and my own observations, I made adjustments and ran each run for 20 episodes to identify if any changes made a significant ie. double improvement in the score. Unfortunately no such step change parameter set was found. Often the score would quickly climb to between 1 and 2 in the first 5 to 8 episodes and then stall for the next 40. I was seeking a solution which demonstrated continuous and regular improvement in the score and reached close to 2 by the 20th episode.

Aside from agent hyper parameter changes, I explored the following structural alterations in the model or agent process.

I explored the following model and agent changes:
- 512/512 networks which started better but then stalled.
- Wider models of 256/128/64 learning did no progress
- Use of Leaky RELUs. This didn’t show an improvement over ReLu and was not used.

- Added batch normalisation after each layer. Looked promising but didn't learn as well as expected
- Added batch normalisation to just the first layer. This worked and was kept
- Reward clipping. This showed and improvement and was kept.
- Forcing weights of local and target networks to same starting values. Learning was very poor
- Removed noise. This worked and showed quicker learning with consistent growth and was kept
- Adjusting the update frequency and samples taken during each update. I tried updating every 20, 10, and 4 episodes with 1 sample or 50% of the episodes (ie 10, 5, 2). In all cases learning didn't progress as well as a single sample update every episode.

From a hyper parameter point of view, I explored the following:
- Using different random seeds. No difference was observed
- Gamma at 0.95. Horrible results
- Batch size to 64. This was best
- Batch size 256. Learning was poor
- Batch size 32. Learning was poor
- Setting learning rate of actor and critic to the same values. Despite reading this working for some people in the forum, it showed a negative impact on learning in my single agent environment
- Setting L2 decay to 0. Learning slowed.

Network architecture

Ultimately, we utilised a two hidden layer model with 400 units in the first and 300 units in the second and used the same model in the actor and final layers of the critic. Batch normalisation was applied in both the actor and critic after a single layer and ReLu activations were used.

No noise was applied during the action step. This was the single biggest improvement of all the parameter and structural changes! A score of 5 was reached in less than 50 episodes with this exclusion with prior runs unlikely to reach 3 by 50 episodes. Without noise a score of 9 was achieved after 68 episodes where as with noise this had previously taken 480 episodes.

The following hyper parameters were used in the final run.

```
BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR_ACTOR = 1e-4         # learning rate of the actor
LR_CRITIC = 3e-4        # learning rate of the critic
WEIGHT_DECAY = 0.0001   # L2 weight decay

UPDATE_EVERY = 1        # timesteps between updates
NUM_UPDATES = 1         # update the agent multiple times sample
```

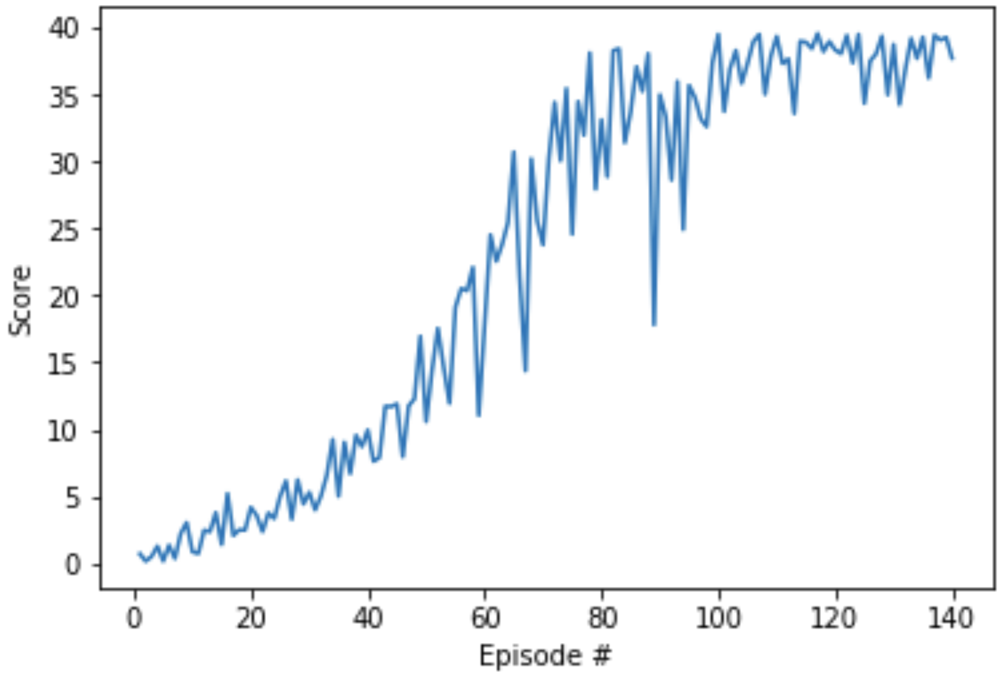
Challenges and learnings

Overall, this project was a challenge due to the hyper parameter setting exploration. While this was frustrating it was ultimately a useful learning process as I became quite familiar with the parameters, what they adjusted and how they impacted learning. The initial bug in the training loop also taught me a lot as I had to manually go through each line of code the provided DDPG example and understand what was happening in each.

One positive outcome of this project has been the learning that even with a general algorithm, there are many parameters which can be tuned and often a large amount of time will be required adjusted, testing and readjusting parameters to get the best result. Frustratingly, what seems to work for one person doesn't necessarily work for others. Hyper parameter setting is definitely an art rather than science.

Results

Using the structure and parameters detail above, the environment was solved in 140 episodes. Below is a chart showing the results from the final run. This appears to be an excellent result.



As can be seen, there was a continual and regular increased in the score, with every episode except one after the first 15 showing an improvement!.

Future Ideas and improvements

During the project I explored many of the suggested adjustments and tuning steps that I had undertaken in prior DNN projects or which had been mentioned in the the forums. Rather than any specific algorithmic or parameter ideas for further exploration I would suggest that the biggest gains could come from improving the tuning and adjustment workflow process.

Tuning is a very manual process, especially when using notebooks, and recording ideas and test runs is often a case of making a manual list, setting parameters, running a test, getting a coffee or snack, checking the results after a bit and then repeating this. There is no version control and no stored history of results.

As I am beginning a new AI project at work this week, I will be aiming to build out some form of version control using git for tracking adjustments and tying the git version id as a key to the output of the test runs and storing this in a simple database.

While the AI algorithms themselves offer plenty of room for experimentation, I believe that the costly human researcher/developer should optimise their process to make best use of this exploration time.