Report: Deep Reinforcement Learning Nanodegree Project 3: Tennis

Michaela G. Lawrence

09 January 2021

1 Introduction

In this environment, two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of `+0.1`. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of `-0.01`. Thus, the goal of each agent is to keep the ball in play.

The observation space consists of `8` variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

The task is episodic, and in order to solve the environment, your agents must get an average score of `+0.5` (over 100 consecutive episodes, after taking the maximum over both agents). Specifically, after each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores.

For this project I chose to use the DDPG algorithm, `ddpg_agent.py` (adapted for multiple agents) and model, `model.py` that were used to solve the double pendulum environment (similar architecture for previous assignment). However, I also made some small but critical changes that are outlined in this report.

One of the changes made was completed in the last project, this included adding two new hyper parameters, `LEARN_EVERY` and `LEARN_NUM`. Another change that I had to make was that I had to account for multiple agents.

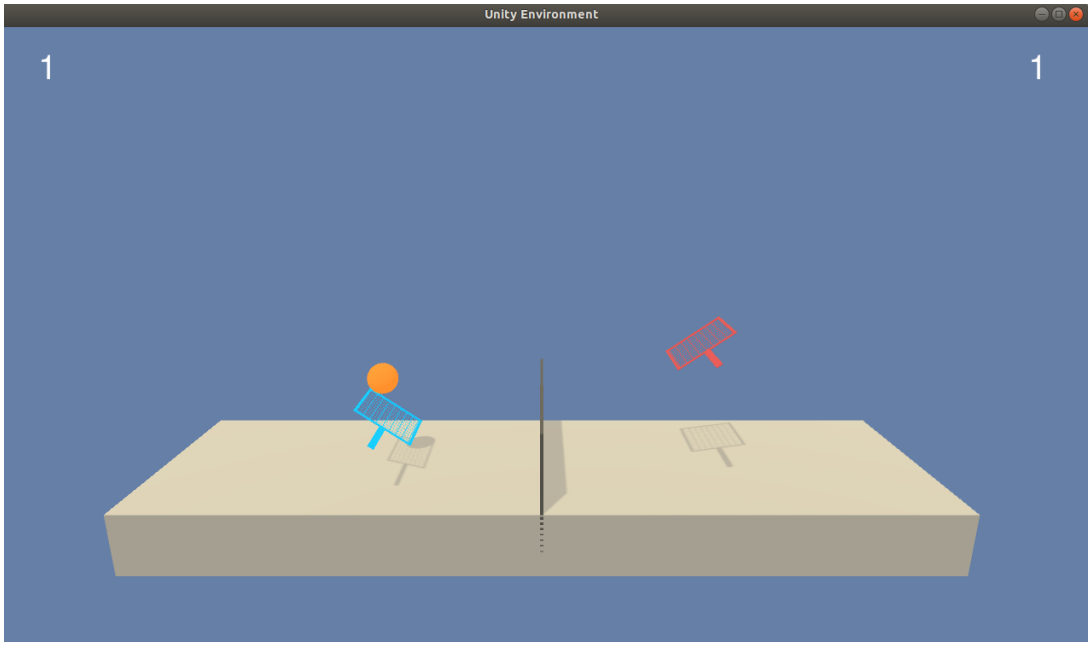

Figure 1: An example of the environment, it contains two tennis rackets, a ball and a net. The tennis rackets are controlled by independent agents and must work collaboratively to rally.

2 Learning algorithm

For this assignment a Deep Deterministic Policy Gradient (DDPG) agent performs episodic training until the environment is solved. Details of how the algorithm works can be found here. The pseudo code can also be found in the aforementioned reference, see Figure 2. However for this assignment there are two agents learning so this algorithm has been adapted to account for multiple agents. Details of how this is done and pseudo code can be found here, in the original paper "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments".

The process can be outlined in four steps

1. The DDPG agent utilises a replay buffer of experiences (the relevant hyper parameter here is `BUFFER_SIZE`). A sample of these experiences (`BATCH_SIZE`) are taken and used to update the neural network.

2. The actor (policy, `LR_ACTOR`) and critic (value `LR_CRITIC`) networks update. The value network is updated similarly as is done in Q-learning. The updated Q value is obtained by the Bellman equation,

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}).$$

   This is where `GAMMA` comes in. The next-state Q values are calculated with the target value network and target policy network. Then, we minimize the mean-squared loss between the updated Q value and the original Q value (which itself is calculated with the target value network). The objective of the policy function is to maximize the expected return. The policy is being updated in an off-policy manner with batches of experience, therefore the mean of the sum of gradients is calculated from the mini-batch and used.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 2: Taken from "Continuous Control With Deep Reinforcement Learning" (Lillicrap et al, 2015)

3. The target network updates by slowly tracking the learned networks via soft updates. This is where `TAU` comes in. We create a copy of the actor and critic networks and call them `actor_target` and `critic_target`. We put the very small value for `TAU`. This means that the target values are constrained to change slowly, improving the stability of learning. This update is performed by function `soft_update`.

4. Exploration is done by adding Ornstein-Uhlenbeck (OU) noise to the action.

In Figure 3 we include the pseudo code for MADDPG.

2.1 The neural networks

classes `Actor` and `Critic` are provided by `model.py`. The typical behavior of the actor and the critic is as follows:

```
actor_target(state) -> action

critic_target(state, action) -> Q-value

actor_local(states) -> actions_pred
```

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

for episode $= 1$ to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for $t = 1$ to max-episode-length **do**
    for each agent i, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \ldots, a_N)$ and observe reward r and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to N **do**
      Sample a random minibatch of S samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma\, Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)\big|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i} \boldsymbol{\mu}_i(o_i^j) \nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$$

    end for
    Update target network parameters for each agent i:

$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$

  end for
end for

Figure 3: Taken from "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments" (Lowe et al, 2017)

`-critic_local(states, actions_pred) -> actor_loss`

The architecture of the neural networks is as follows:

3 fully-connected layers and 2 rectified nonlinear layers. The number of neurons of the fully-connected layers are as follows:

for the actor:
Layer fc1, number of neurons: state_size × fc1_units,
Layer fc2, number of neurons: fc1_units × fc2_units,
Layer fc3, number of neurons: fc2_units × action_size,

for the critic:
Layer fcs1, number of neurons: state_size × fcs1_units,
Layer fc2, number of neurons: (fcs1_units+action_size) × fc2_units,
Layer fc3, number of neurons: fc2_units × 1.

3 Hyper parameters

- `BUFFER_SIZE` (int): `1e5` replay buffer size (Rediced from value for the previous assignment to decrease the amount of time that my code was taking)
- `BATCH_SIZE` (int): `512` minibatch size (four times the size used in the previous assignment, with this setting my agent learned in fewer episodes and learning was more stable)
- `GAMMA` (float): `0.99` discount factor
- `TAU` (int): `1e-3` for soft update of target parameters
- `LR_ACTOR` (float): `1e-3` learning rate for the actor
- `LR_CRITIC` (float): `1e-3` learning rate for the critic
- `WEIGHT_DECAY` (float): `0` L2 Weight decay `LEARN_EVERY` (int): `32` learning timestep interval `LEARN_NUM` (int): `10` number of learning passes

How did I decide on these values for the hyper parameters? Something that I found more prevalent in this assignment than the previous two was trying to get the agents to learn in a stable fashion. This can be seen in the final results as well, see Figure 4. The total average over the last 100 episodes does not climb smoothly as they did for the previous two assignments, it goes up and down.

The hyper parameters are based on the values found in `assignment two` however some changes have been made. I chose to decrease `BUFFER_SIZE` by an order of magnitude because this decreased the amount of time the algorithm took to run on my computer (because it stores less in memory).

I then left all other hyper parameters and ran the code to see how it was doing. I saw that learning was somewhat unstable. The rolling average could take just a few hundred episodes to reach values above `+0.4` but then this average would fall and climb and fall again, not reaching the threshold for solution of `+0.5`. I first doubled `BATCH_SIZE` to 256 and found learning to be slightly more stable and would eventually reach the threshold. I was still uneasy about the instability at this point so I doubled the `BATCH_SIZE` again. As you can see from the final results the learning is still unstable but is better than when the batch size was smaller.

I didn't play around with the other hyper parameters, so they remain the same as they did for the previous assignment but this is something that could be done to try to further improve performance.

4 Plot of rewards

The best results can be found in Figure 4.

```
Episode 100 Total Average Score:  -0.002
Episode 200 Total Average Score:  -0.005
Episode 300 Total Average Score:  -0.002
Episode 400 Total Average Score:  0.011
Episode 500 Total Average Score:  0.037
Episode 600 Total Average Score:  0.037
Episode 700 Total Average Score:  0.075
Episode 800 Total Average Score:  0.144
Episode 900 Total Average Score:  0.235
Episode 1000 Total Average Score:  0.358
Episode 1100 Total Average Score:  0.409
Episode 1200 Total Average Score:  0.445
Episode 1300 Total Average Score:  0.432
Problem Solved after 1317 episodes!  Total Average score:  0.503
```

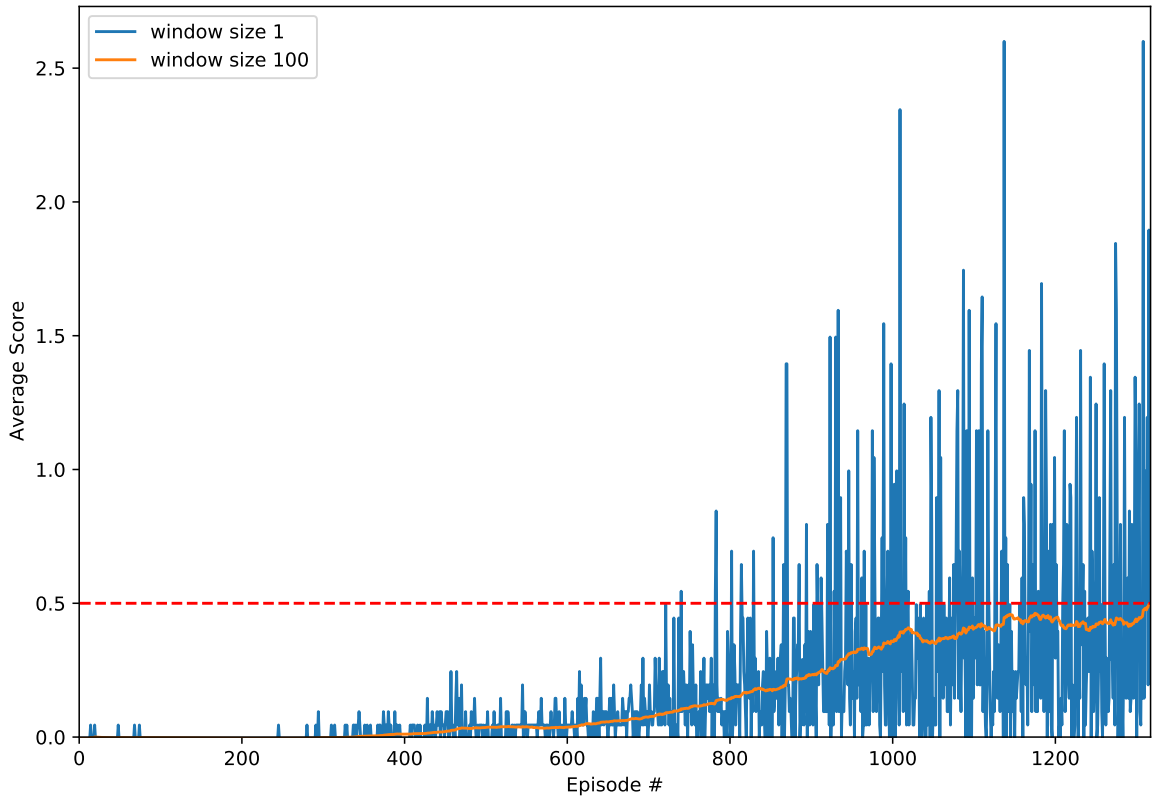

Figure 4: The best results.

5 Ideas for future work

Automated hyper parameter tuning:
I spent a lot of time tweaking hyper parameters. An automated hyper parameter tuning process could be used to find the best settings in an automated fashion. Hyper parameters that could benefit from this could be: `BUFFER_SIZE`, `BATCH_SIZE`, `LR_ACTOR`, `LR_CRITIC`, `LEARN_EVERY`, `LEARN_NUM`.

Additionally to these parameters I could consider adjusting the OU noise parameters that can be found on line `153` of `dppg_agent_multi.py`

NN architecture:
Its a possibility that performance could improve by changing the number of layers in the neural networks. Some papers state that batch normalization can accelerate deep network training, for example see, here and here.

Alternative methods:
I could try a different method such as Twin Delayed DDPG for multiple agents.