Report: Deep Reinforcement Learning Nanodegree Project 2: Arms

Michaela G. Lawrence

05 January 2021

1 Introduction

In the environment, double-jointed arms can move to target locations. A reward of `+0.1` is provided for each step that an agent's hand is in the goal location. Thus, the goal of an agent is to maintain it's position at the target location for as many time steps as possible. The version of the environment that we solve here contains 20 identical agents, each with its own copy of the environment. See Figure 1 for an example of what the environment could look like.

The environment is considered solved when agents achieve an average score of `+30` (over 100 consecutive episodes, and over all agents).

The observation space consists of 33 variables (per arm) corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between `-1` and `+1`.

This version of the task where 20 arms are active at once lends itself to be solved using distributed training. Algorithms such as PPO, A3C, and D4PG use multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience.

For this project I chose to use the DDPG algorithm, `ddpg_agent.py` and model, `model.py` that were used to solve the double pendulum environment. However, I also made some small but critical changes that are outlined in this report.

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.

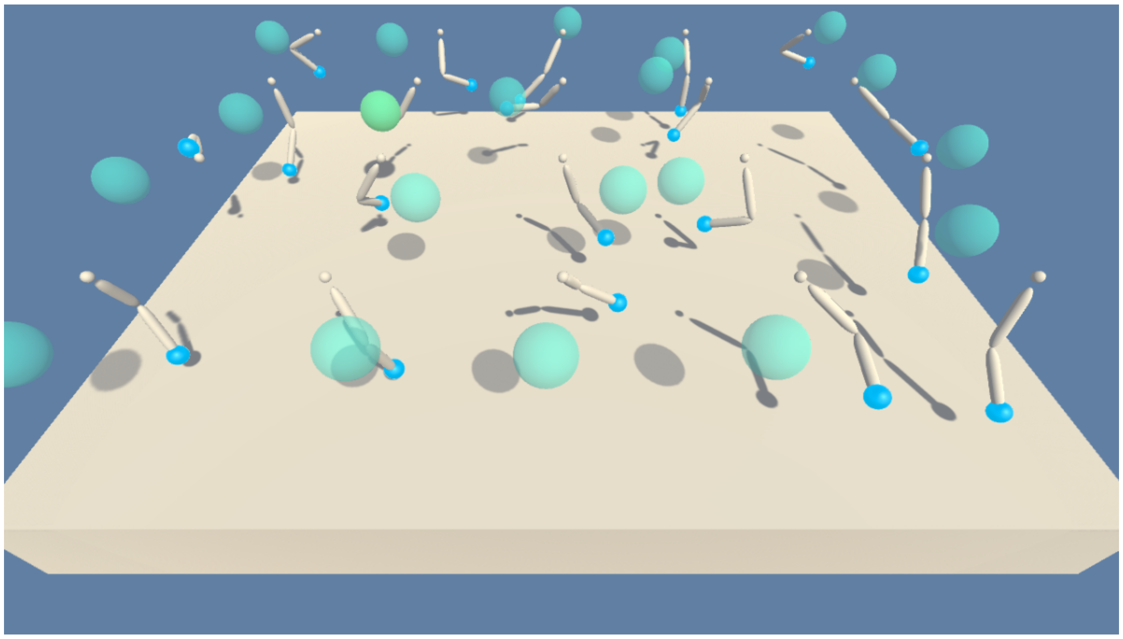


Figure 1: An example of the environment, it is filled with twenty (randomly acting) arms. Each arm is fixed at the "shoulder". The goal location of the bright blue "hands" at the end of the arms are the translucent blue location spheres. In the top left of the image, one of the hands at the end of an arm has entered the goal location and so the location sphere has changed colour to solid green.

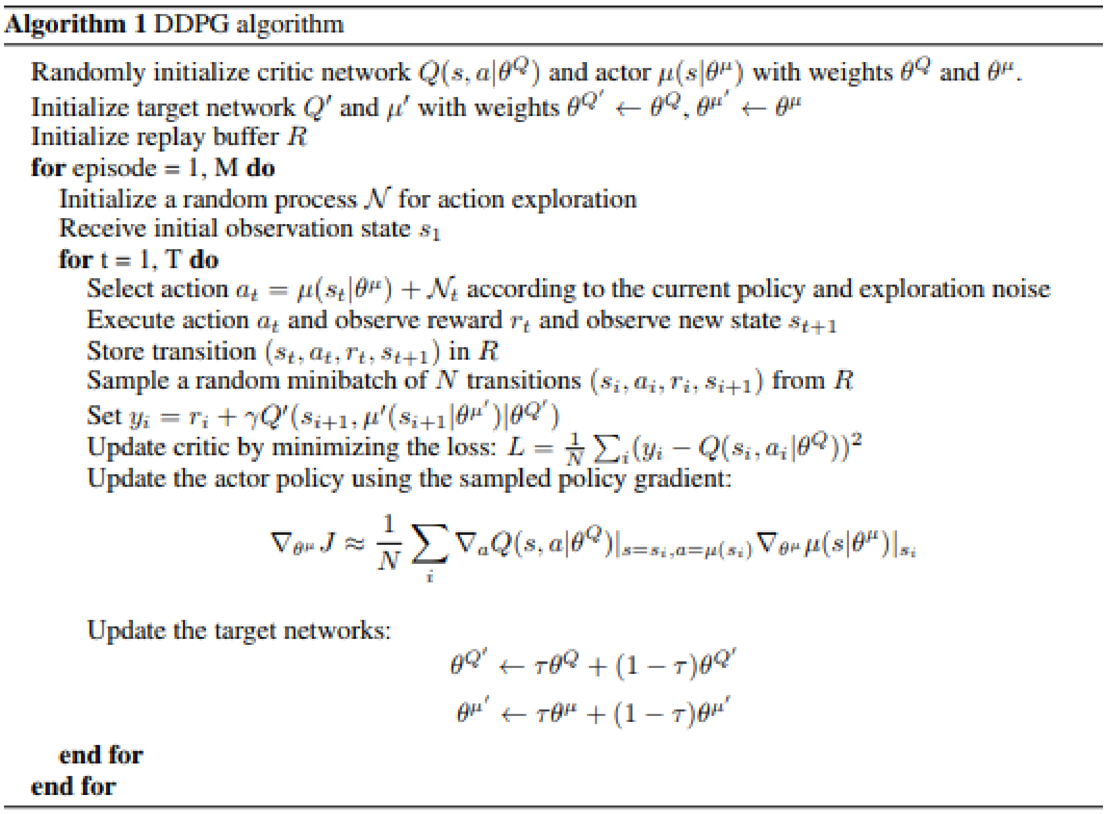
The process can be outlined in four steps

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.

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 noise to the action.



```
Algorithm 1 DDPG algorithm
  Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
  Initialize target network $Q'$ and $\mu'$ 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)

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

-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 $\times$ fc1_units,
Layer fc2, number of neurons: fc1_units $\times$ 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): `1e6` replay buffer size
- `BATCH_SIZE` (int): `128` minibatch size
- `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? The hyper parameters listed with the dots, •, are based on the values found in `ddpg_agent.py` however I chose to increase `BUFFER_SIZE` by an order of magnitude from the start as it was recommended in the student chat. The other, •, parameters were decided by changing them slightly up and down and looking at the results after `60` episodes and this combination was found to be the best.

I then ran the code `ddpg_agent.py` with these, •, hyper parameters for a much longer time and found that the highest average reward never got a higher score than around two. This is where the two additional hyper parameters ∗ come in. After looking for help on the student Q&A forum, I found that it was essential to make a change to the code. These changes can be found in these chosen hyper parameters and around line 60 in the agent file. It was recommended to try `LEARN_EVERY` = 20 and `LEARN_NUM` = 10. This is what I did to start with and found an average score of around 20 after `120` episodes. I then increased `LEARN_EVERY` to 32 and found that my code ran more quickly and achieved the goal score in only 107 episodes.

4 Plot of rewards

The best results can be found in Figure 3.

```
Episode 10 Total Average Score:  2.75
Episode 20 Total Average Score:  5.76
Episode 30 Total Average Score:  9.64
Episode 40 Total Average Score:  13.70
Episode 50 Total Average Score:  17.53
Episode 60 Total Average Score:  20.78
Episode 70 Total Average Score:  23.25
Episode 80 Total Average Score:  25.10
Episode 90 Total Average Score:  26.53
Episode 100 Total Average Score:  27.63
Ep 107 Total Avg Score:  30.12 Mean:  37.19 Min:  31.60 Max:  39.30 Duration:  85.38
Problem Solved after 107 episodes!  Total Average score:  30.12
```

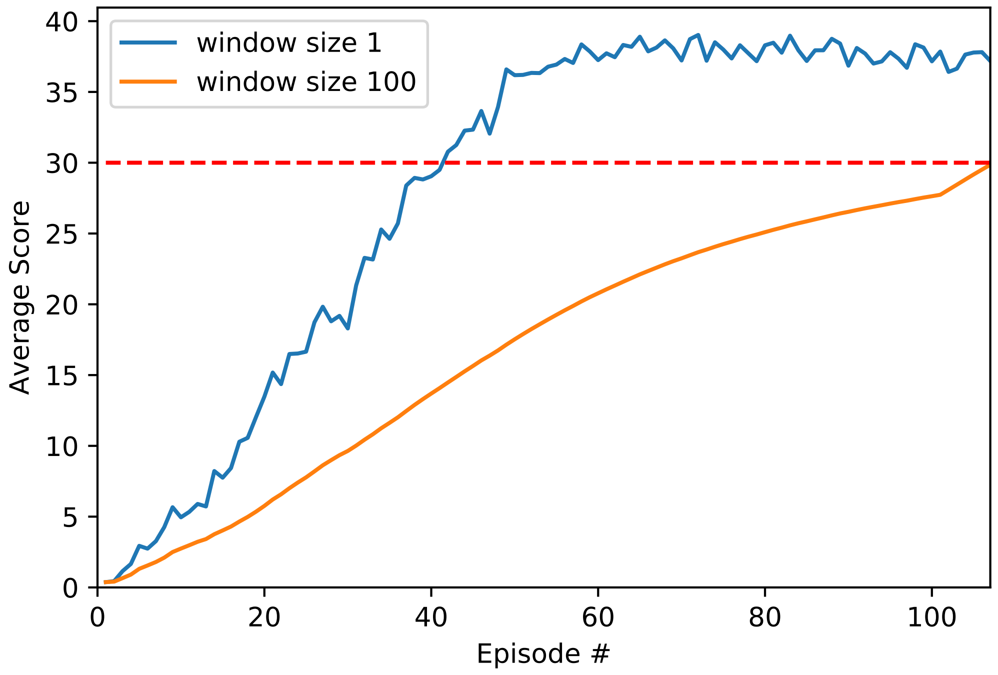


Figure 3: The best results.

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: `BATCH_SIZE`, `LR_ACTOR`, `LR_CRITIC`, `LEARN_EVERY`, `LEARN_NUM`.

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:
Instead of DDPG, other models can be considered, such as PPO, A3C and others. Originally, I had intended to try to solve the problem with A3C but ran into some IT issues.