Project Continuous Control

Luís Melo dos Santos

July 21, 2019

This is the second project in a series of projects linked to the *Deep Reinforcement Learning for Enterprise Nanodegree Program* at *Udacity*.

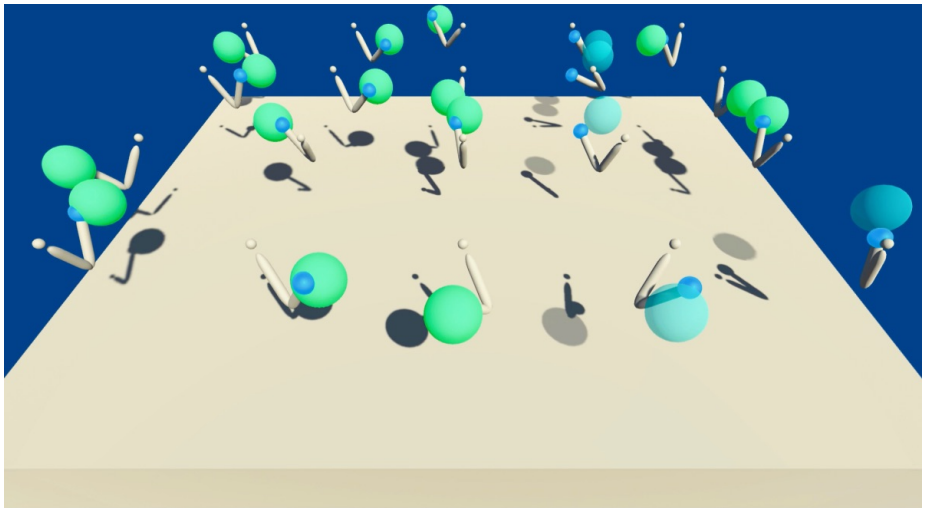
In this project, we aim to train agents in a set of `Reacher` *Unity* environments.[1]

In these environments the agents are double-jointed arms surrounded by target locations. A reward of +0.1 is provided for each step that the agent's hand is in the target location. Thus, the goal of the agent is to maintain its position at the target location for as many time steps as possible.

The observation space consists of 33 variables 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.

There are two distinct `Reacher` *Unity* environments used in this project,

[1] These environments were tailored by Udacity specifically for this nanodegree, similar to the original `Reacher` environment publicly available in the *Unity* github repository.



- *One Agent* - where there are only one robotic arm and one target in the environment;
- *Twenty Agents* - where there are twenty distinct robotic arms and twenty different targets in the environment.

The task is episodic, and in order to solve the *One Agent* environment, the agent must get an average score of +30 over 100 consecutive episodes. For the *Twenty Agents* case, the environment is considered solved when the average score of the twenty agents over 100 consecutive episodes reaches also +30.

1 Learning Algorithm

We started the project by focusing on the *One Agent* environment case and trying to get as much insight from solving this simpler environment.

1.1 *One Agent* environment

For the *One Agent* environment we adapted the algorithm used in class to solve the `Pendulum-v0` *OpenAI* environment. This algorithm is inspired in the Deep Deterministic Policy Gradient (DDPG) model introduced in Lillicrap *et al.* paper [1] in 2016. In the article the authors propose to adapt the ideas underlying the success of the DQN model [2] to the continuous action domain.

From the DQN model we continue to use a storage of previous experiences - *ReplayBuffer()* - randomly accessed by the agent for the learning process, in an attempt to economically maximise information absorption from past experiences and stabilising learning in the process.

For stability, we also keep the concept of local and target neural networks. In the learning process the local neural networks with weights θ learn from experiences but the target neural networks with weights θ' are kept constant until a separate step - *soft_update()* - when θ' gets updated according to the equation,

$$\theta' = \theta' + \tau(\theta - \theta'), \tag{1}$$

where τ is the *soft_update()* learning rate parameter controlling the rate at which new information propagates from the local network through to the target neural network.

The DDPG model departs from the DQN model by taking some inspiration from actor-critic models. It tries to both approximate a policy μ and a value function Q with neural networks.

The policy neural network μ,

$$a = \mu(s|\theta^\mu) \tag{2}$$

maps from each state s in the environment a single continuous action a using the neural network weights θ^μ as the control parameters of the approximation -

variables that the learning algorithm will vary in order to improve the approximation.

The Q neural network will have a slightly altered structure from the one in the DQN model in order to accommodate continuous actions. This time around Q,

$$v = Q(s, a|\theta^Q) \tag{3}$$

will have both the state of the environment s and action a as inputs and will only provide a real number v as output - the value of the respective state s and action a in the environment. Again, θ^Q correspond to the weights of the neural network that we will use to control the quality of the approximation.

The Q-learning in DDPG is similar to the Q-learning in DQN and based on a classical 1-step temporal-difference method – TD(0) – introduced in [3] where a $Q(S, A)$ state-action value function gets updated by ΔQ at each step with new information $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ acquired by the agent as it interacts with the environment,

$$\Delta Q(S_t, A_t) = \alpha\big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big] \tag{4}$$

S_t is the current state that the agent occupies in the environment, A_t is the action that the agent takes, R_{t+1} is the reward that the agent receives from the environment, S_{t+1} is the state where the agent ends up after taking the action and A_{t+1} is the next action the agent will take at state S_{t+1}. The remaining two parameters in the *Q-update* equation (4) are,

- γ – the discount rate of future rewards;
- α – the learning rate.

γ dictates how eager our agent is to defer rewards and α controls the speed at which the new information transforms the Q function.

In the context of the DDPG model equation (4) takes the form,

$$\Delta Q(S_t, A_t|\theta^Q) = \alpha\big[R_{t+1} + \gamma Q'(S_{t+1}, \mu'(S_t|\theta^{\mu'})|\theta^{Q'}) - Q(S_t, A_t|\theta^Q)\big] \tag{5}$$

where prime refers to the target versions of both policy μ and value Q neural networks.

A_t will be determined not only by the policy network μ but also by a noise factor $\mathcal{N}$ during the learning process,

$$A_t = \mu(S_t|\theta^\mu) + \mathcal{N} \tag{6}$$

so the agent can keep exploring, making DDPG an off-policy learning algorithm.

In the original paper [1] the noise factor $\mathcal{N}$ was chosen to be a mean-reverting Ornstein-Uhlenbeck process,

$$dx = -\theta * x + \sigma * U(1) \tag{7}$$

where $U(1)$ is the unity uniform distribution. In our model we will be reducing exponentially the noise factor as learning progresses.

The update of the value function Q in equation (5) can be codified as a gradient descent step on minimising the loss function L^Q with the following structure,

$$L^Q(\theta^Q) = \frac{1}{N}\sum_i \left[R_{t+1} + \gamma Q'(S_{t+1}, \mu'(S_t|\theta^{\mu'})|\theta^{Q'}) - Q(S_t, A_t|\theta^Q)\right]^2 \tag{8}$$

where N is the number of experiences in each learning batch.

The policy μ update can also be codified as a gradient descent of the loss function L^μ,

$$L^\mu(\theta^\mu) = -\frac{1}{N}\sum_i Q(S_t, \mu(S_t, \theta^\mu)|\theta^Q) \tag{9}$$

making the μ policy the greedy policy solution of the environment.

The neural network structure we chose for both actor μ and critic Q approximations was again based in article [1].

For the policy μ we have used a fully connected neural network with two hidden layers. The input layer has the size of the state space and the output layer has the size of the action space. We used $tanh$ as the last layer transformation in order to keep action values inside the interval $[-1, 1]$.

For the value function Q we have also used a fully connected neural network with two hidden layers. The input layer has again the size of the state space. It is only in the second hidden layer that we introduce the action values concatenated to the values coming from the first hidden layer. The output layer has only one node with no non-linear transformation performed. We use ReLU functions for any other node transformations and Adam was kept as the optimiser of choice.

1.2 *Twenty Agents* environment

Only a few details changed when we worked in the *Twenty Agents* environment. We adapted the DDPG model to the *Twenty Agents* environment based on Barth-Maron *et al.* article [4] from 2018, transforming it into a Distributed Deep Deterministic Policy Gradient (D3PG) model.

In our implementation we distributed the same policy μ and value function Q in synchronous between all agents in the environment. All experiences were shared between the agents as agents updated and learnt from the same memory pool.

As recommended in [4] we simplified the noise factor $\mathcal{N}$ to a random variable from a standard normal distribution,

$$\mathcal{N} = \epsilon N(0, 1) \tag{10}$$

This way we have reduced the number of noise hyperparmeters to one, ϵ. We still kept the same exponential noise reduction process as learning progresses from the previous environment.

In the next section, we will detail how we implemented these learning algorithms in practice.

2 Implementation

Our implementation of the learning algorithms, as mention previously, follows closely the code architecture introduced in class. The definition of the agents is split between two files,

- `ddpg/d3pg_agent_final.py`
- `model_ddpg/d3pg_final.py`

In `ddpg/d3pg_agent_final.py`, we define the *Agent()* class. One of the alterations to the class definition we introduce was to expand the inputs and include a dictionary with all the hyperparameters to improve flexibility in analysing and tuning the model. In this class the methods *step*, *act*, *learn* and *soft_update* get defined. In *step()* we sequence how the agent stores each experience in memory and how often it starts the learning process. In *act()*, we define the *behaviour policy* of the agent – based on equation (6) if it is learning or the greedy policy μ otherwise. It is in the *learn()* method that the loss functions L^Q and L^μ, as defined in (8, 9), get calculated and optimised. In this implementation we use an Adam optimization.[2] It is also worth mentioning that we perform batch learning – for each training step we sample the agent's memory for a set of experiences and use their information in the optimisation step. At each learning step we also perform a soft update of the target neural network parameters $(\theta^{Q'}, \theta^{\mu'})$, according to equation (1), using the *soft_update()* method.

A second class is defined in `ddpg/d3pg_agent_final.py` – the *ReplayBuffer()* class – where the agent's memory structure gets defined. In this case a *queue* of fixed maximum size is initialised and *add()* and *sample()* methods get defined. In the DDPG and D3PG models, memory sampling is done randomly where each experience has an equal probability[3] to be picked for the learning process.

In `model_ddpg/d3pg_final.py`, the neural networks get defined. We use *PyTorch* to define a fully connected deep neural network with two hidden layers.[4] Both input layer units and output layer units are fixed in the DDPG and D3PG models but we kept the number of units for both hidden layers variable in order to be able to test and tune them.

We created a third file `training_ddpg/d3pg_agent_final.py` in order to systematise our agents' training. We loop the agents through episodes keeping track of their performance in order to establish when they have successfully solved the environment. We also define a maximum amount of episodes `n_episodes` after which if our agents are still not successful we stop the training process.

[2] We kept Adam optimisation from the original implementation and it was something we have not tested in terms of performance against other optimisation schemes.

[3] with replacement.

[4] We have not tested the performance of our neural network architecture against other types

Name	Article	Tuned	Description
n_episodes	500	500	max no of episodes per training session
max_t	1000	1000	max no of steps per episode
actor_fc1_units	400	400	no of units for first hidden layer actor net
actor_fc2_units	300	300	no of units for second hidden layer actor net
critic_fc1_units	400	400	no of units for first hidden layer critic net
critic_fc2_units	300	300	no of units for second hidden layer critic net
buffer_size	10^6	10^5	memory size
batch_size	64	128	no of examples fed at each learning step
gamma	0.99	0.99	rewards discounting rate
tau	10^{-3}	10^{-3}	soft update learning rate for target nets
lr_actor	10^{-4}	10^{-4}	local actor net learning rate
lr_critic	10^{-3}	10^{-4}	local critic net learning rate
weight_decay	10^{-2}	0	L2 weight decay
update_every	1	1	no of action steps per learning step
noise_theta	0.15	0.3	Ornstein-Uhlenbeck process θ parameter
noise_sigma	0.2	0.2	Ornstein-Uhlenbeck process σ parameter

Table 1: Hyperparameters values used in the DDPG analysis.

3 Results

In this section we present the results of our analysis for both *One Agent* and *Twenty Agents* environments. For each environment we produced a notebook that details the steps of our analysis.

3.1 DDPG analysis

In the *One Agent* environment we implemented the DDPG model according to Lillicrap *et al.* article [1] including the hyperparameters values mentioned in the *Experiment Details* section.

The hyperparameters used are presented in the column named *Article* in Table 1. The agents were not able to make significant progress in learning the task using this particular hyperparameter selection, as we can see in the graph in Figure 1.

We proceeded in exploring other combinations of hyperparameters that would help improve the performance of the agents. Also in Table 1 in the column named *Tuned* we present the combination of hyperparameters that we found to best optimise the learning ability of the agents in the *One Agent* environment.

In the graph in Figure 2 we present the agent's learning performance on 5 different training sessions. Using this tuned hyperparameters combination the agents were able to solve the environment, on average just under 200 episodes with a standard deviation of 50 episodes which reflects good enough stability.

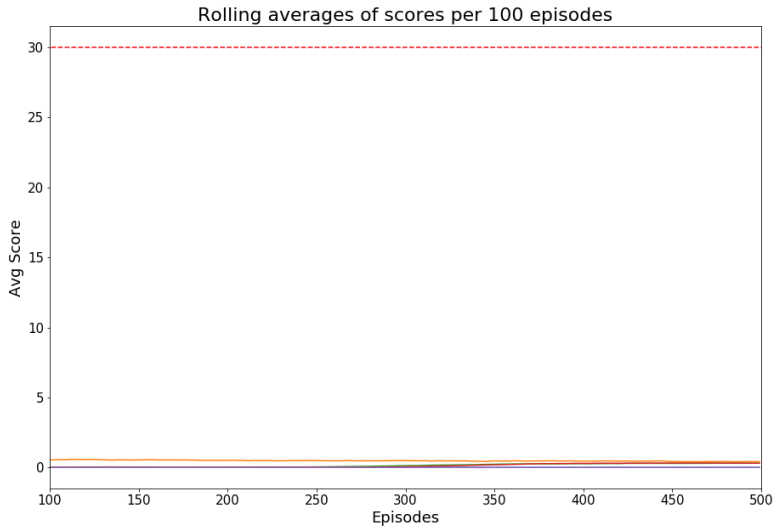


Figure 1: Rolling average scores of 5 different training sessions using Lillicrap *et al.* article [1] hyperparameters on a DDPG model.

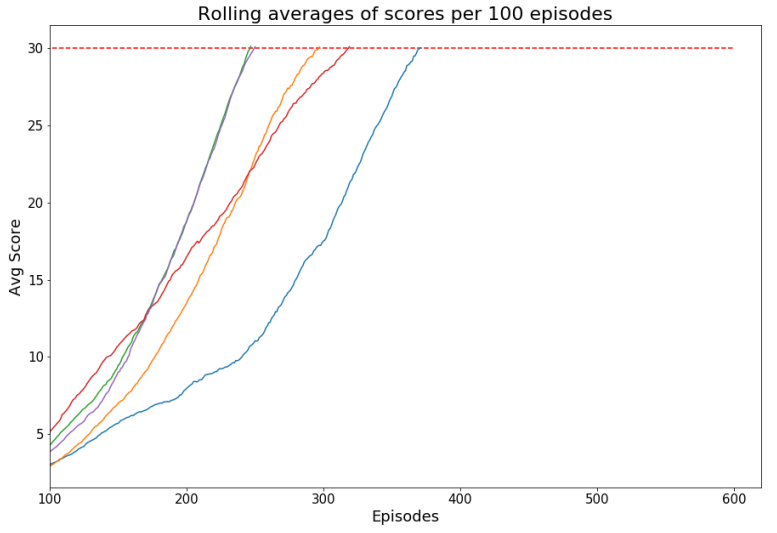


Figure 2: Rolling average scores of 5 different training sessions using tuned hyperparameters on a DDPG model.

Name	Article	Tuned	Description
n_episodes	500	200	max no of episodes per training session
max_t	1000	1000	max no of steps per episode
actor_fc1_units	400	400	no of units for first hidden layer actor net
actor_fc2_units	300	300	no of units for second hidden layer actor net
critic_fc1_units	400	400	no of units for first hidden layer critic net
critic_fc2_units	300	300	no of units for second hidden layer critic net
buffer_size	10^6	10^5	memory size
batch_size	256	128	no of examples fed at each learning step
gamma	0.99	0.99	rewards discounting rate
tau	10^{-3}	10^{-3}	soft update learning rate for target nets
lr_actor	10^{-4}	10^{-4}	local actor net learning rate
lr_critic	10^{-4}	10^{-4}	local critic net learning rate
weight_decay	0	0	L2 weight decay
update_every	1	1	no of action steps per learning step
epsilon	0.3	0.3	uniform noise process initial scale parameter

Table 2: Hyperparameters values used in the D3PG analysis.

3.2 D3PG analysis

In the *Twenty Agents* environment we implemented the D3PG model according to Barth-Maron *et al.* article [4] including the hyperparameters values mentioned in the *Results* section.

The hyperparameters values are presented in the column named *Article* in Table 2. The agents were able to solve the environment significantly quicker than in the *One Agent* environment even with the tuned DDPG model, as we can see in the graph in Figure 3, but not always. In some training sessions the model failed to make any progress in the 500 episodes limit we imposed. The model showed a high level of instability.

We proceeded by changing the hyperparameters values combination to the one found for the DDPG model in the *One Agent* environment. The hyperparameter values used are in Table 2 in the column named *Tuned*. That turned out to be a suitable combination for the *Twenty Agents* environment as well.

As we can see in the graph in Figure 4, the agent's learning performance on 5 different training sessions looks quite stable and fast. Using this tuned hyperparameters combination the agents were able to solve the environment, on average just under 25 episodes with a standard deviation of 6.5 episodes which reflects good combination of speed and stability.

In terms of time, the overhead of controlling and processing information for 20 agents - going from an average of 16 seconds per episode in our machine for one agent to 20 seconds for twenty agents - greatly pays off in the number of episodes needed to solve the environments - going from 200 episodes for one agent to 25 episodes on average.

Figure 3: Average scores of 5 different training sessions using Barth-Maron *et al.* article [4] hyperparameters on a D3PG model.

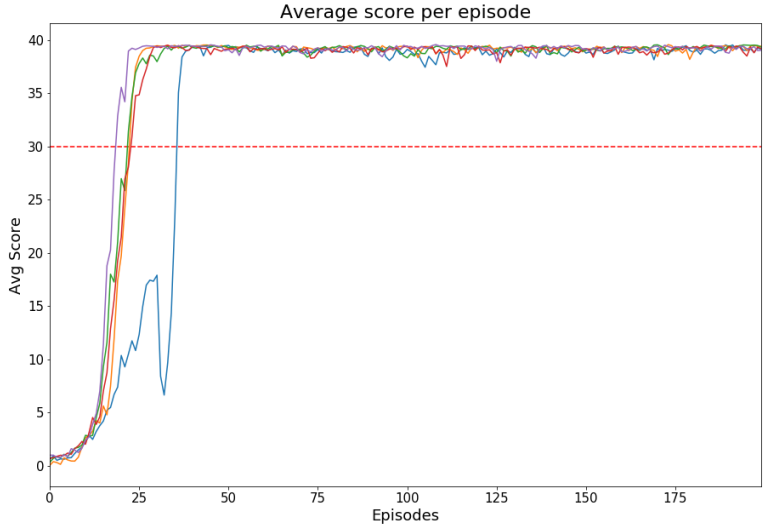


Figure 4: Average scores of 5 different training sessions using tuned hyperparameters on a D3PG model.

4 Ideas for Future Work

If we would have not been able to solve the environment or were not satisfied with the performance achieved by the model so far in this project, there would have been a few natural and immediate steps we could have taken to enhance model performance.

From the paper that we have used for our D3PG model implementation [4], we get the advice that moving from TD(0) to TD(4) - picking up 5-step returns in our trajectories rather than just one as we currently do in our model with standard temporal differences (5) - improves performance and stability of the model.

They have also mentioned that prioritising experiences when sampling the memory can also boost performance in most cases. And, obviously we could also explore the idea of implementing a D4PG model by upgrading our value function Q to a distribution Z where Q is just the expectation value [4, 5],

$$Q_\mu(s,a) = \mathbb{E}(Z_\mu(s,a)) \tag{11}$$

One of the tips from the course that we have also left behind was to explore the structure of the noise factor $\mathcal{N}$ in (6). It seems that if we introduce noise directly into the actor's neural network weights θ^μ - NoisyNets [6] - the model produces more 'useful' noise and improves learning performance and stability.

Some of these enhancements might come handy as we intend to tackle next the `Crawler` - a more challenging *Unity*'s environment as suggested in the lecture notes.

There is a wealth of models in the continuous control space out there [7, 8, 9] that we can tap into for inspiration and improvement.

Our idea is to keep this project going and continue posting more notebooks with further improvements and more results of our exploration as we go along.

Watch this space... ☺

References

[1] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," *arxiv.org*, February 2016.

[2] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, 2015.

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning - An Introduction*. The MIT Press, second ed., 2018.

[4] G. Barth-Maron *et al.*, "Distributed distributional deterministic policy gradients," *ICLR conference*, 2018.

[5] M. G. Bellemare *et al.*, "A distributional perspective on reinforcement learning," *arxiv.org*, no. 1707.06887, 2017.

[6] M. Fortunato *et al.*, "Noisy networks for exploration," *arxiv.org*, no. 1706.10295, 2017.

[7] Y. Duan *et al.*, "Benchmarking deep reinforcement learning for continuous control," *arxiv.org*, no. 1604.06778, 2016.

[8] J. Schulman *et al.*, "Proximal policy optimization algorithms," *arxiv.org*, no. 1707.06347, 2017.

[9] V. Mnih *et al.*, "Asynchronous methods for deep reinforcement learning," *arxiv.org*, no. 1602.01783, 2016.