Robot Arm Navigation using Deep Deterministic Policy Gradients (DDPG)

By

Wael Farag

1. Goal

In this project, the Reacher environment [1] is used to emulate the movement of a double-jointed robot arm that can move to target locations. The main goal of this project is to design and implement an Agent to control the movement this double-jointed robot arm. The design and implementation should use one of the Policy-Based Methods [2] or the subcategory policy gradient methods [3]. Examples like REINFORCE Algorithm [4], Proximal Policy Optimization (PPO) algorithms [5], Asynchronous Advantage Actor-Critic (A3C) algorithms [6], Advantage Actor-Critic (A2C) algorithms [7], Generalized Advantage Estimation (GAE) algorithms [8], Trust Region Policy Optimization (TRPO) algorithm [9], Truncated Natural Policy Gradient (TNPG) algorithm [9], Deep Deterministic Policy Gradient (DDPG) algorithms [9][10], and Distributed Distributional Deterministic Policy Gradients (D4PG) [11].

A reward of +0.1 is provided for each step that the agent's hand is in the goal 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. The task is episodic, and in order to solve the environment, the agent must collect rewards and aggregate them for the latest 100 consecutive episodes. The environment is considered solved, when the average (over 100 episodes) of the agent score is at least +30.

2. Approach

There are two versions of the environment:

1. The First version is the single arm one. This environment is built by unity technologies [1] and can be downloaded from this link. This environment will be solved using a single agent trained using a designed Deep Deterministic Policy Gradient (DDPG) algorithm [9]. The training task is episodic, and in order to solve this environment, the agent must get an average score of +30 over 100 consecutive episodes.

2. The Second version is the 20 arm one. This environment is built by unity technologies [1] and can be downloaded from this link. This environment will be solved using 20 agents trained simultaneously using a designed Distributed Distributional Deterministic Policy Gradients (D4PG) algorithm [9]. The barrier for solving this environment is slightly different, to take into account the presence of many agents (20). In particular, the agents must get an average score of +30 (over 100 consecutive episodes, and over all agents). After each episode, the rewards are added up so that each agent received (without discounting), to get a score for each agent. This yields 20 (potentially different) scores. Then the average of these 20 scores is taken. This yields an average score for each episode (where the average is over all 20 agents).

3. Evaluating the State-Action Space

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the robot 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.

In other words, the *state space* for this project consists of 11 continuous 3-dimensional vectors representing the position, rotation, velocity, and angular velocities of the two parts of a virtual "Reacher" arm, along with the position of the "hand" and the position and speed of the goal sphere. The goal sphere is programmed to circle around the arm (within the X-Y plane, Z is constant).

The *action space* is a vector with four numbers, clamped between -1 and 1, corresponding to the X and Z torques applicable to arm's two joints ("shoulder" and "elbow"). Code for the **Reacher**

Agent can be viewed here. Figure 1 below shows a version of the environment with 10 arms follow the goal spheres movements.

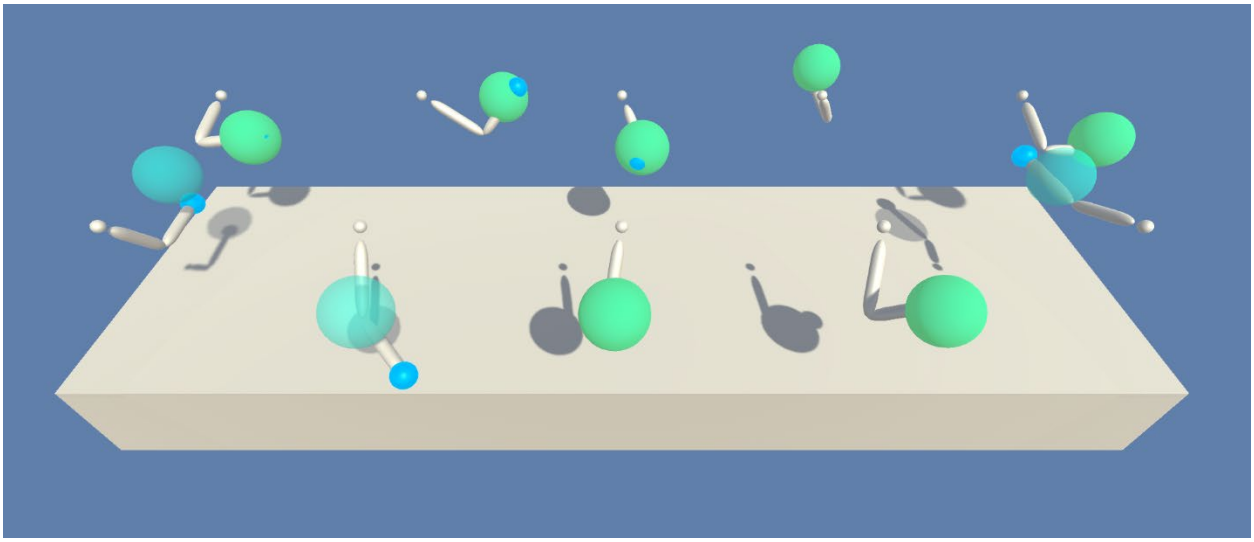

Figure 1 Multiple-Arm Reacher Environment.

4. Implementing the Learning Algorithm

DDPG uses four neural networks [12]: a Q network, a deterministic policy network, a target Q network, and a target policy network.

$$\text{Parameters:}$$
$$\theta^{Q} : \text{Q network}$$
$$\theta^{\mu} : \text{Deterministic policy function}$$
$$\theta^{Q'} : \text{target Q network}$$
$$\theta^{\mu'} : \text{target policy network}$$

The Q network and policy network is very much like simple Advantage Actor-Critic, but in DDPG, the Actor directly maps states to actions (the output of the network is directly the agent output) instead of outputting the probability distribution across a discrete action space.

The target networks are time-delayed copies of their original networks that slowly track the learned networks. Using these target value networks greatly improve stability in learning. Here's why: In methods that do not use target networks, the update equations of the network are

interdependent on the values calculated by the network itself, which makes it prone to divergence. For example:

This depends Q function itself (at the moment it is being optimized)

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a) + \gamma max\overbrace{Q(s',a')} - Q(s,a)]$$

So, the standard Actor & Critic architecture for the deterministic policy network and the Q network is usually used.

The pseudo-code learning algorithm for the 4 networks is shown in Figure 2, and it has been implemented in this repository.

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 The pseudo-code for the DDPG algorithm.

DDPG uses a Replay Buffer [13] to sample experience to update the 4 neural network parameters. During each trajectory roll-out, all the experience tuples (state, action, reward, next_state) are saved and stored in a finite-sized cache — a "replay buffer." Then, we sample random mini-batches of experience from the replay buffer when we update the value and policy networks.

The value network is updated similarly as is done in Q-learning. The updated Q value is obtained by the Bellman equation:

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

However, in DDPG, the next-state Q values are calculated with the target value network and target policy network. Then, the mean-squared loss between the updated Q value and the original Q value is minimized as follows:

$$Loss = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$$

For the policy function, the objective is to maximize the expected return:

$$J(\theta) = \mathbb{E}[Q(s,a)|_{s=s_t, a_t=\mu(s_t)}]$$

To calculate the policy loss, we take the derivative of the objective function with respect to the policy parameter. Keeping in mind that the actor (policy) function is differentiable, so the chain rule ought to be applied.

$$\nabla_{\theta^\mu} J(\theta) \approx \nabla_a Q(s,a) \nabla_{\theta^\mu} \mu(s|\theta^\mu)$$

But since the policy is updated in an off-policy way with batches of experience, we take the mean of the sum of gradients calculated from the mini-batch:

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

A copy of the target network parameters is made and have them slowly track those of the learned networks via "soft updates," as illustrated below:

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

$$\text{where} \quad \tau \ll 1$$

In Reinforcement learning for discrete action spaces, exploration is done via probabilistically selecting a random action (such as epsilon-greedy or Boltzmann exploration). For continuous action spaces, exploration is done via adding noise to the action itself. In the DDPG paper [9], the authors use Ornstein-Uhlenbeck Process to add noise to the action output (Uhlenbeck & Ornstein, 1930) [14]:

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N}$$

The *Ornstein-Uhlenbeck Process* generates noise that is correlated with the previous noise, as to prevent the noise from canceling out or "freezing" the overall dynamics [15].

5. Experimentations and Results

The Deep Deterministic Policy Gradient (DDPG) algorithm is implemented for both the "Single Arm" and the "20-Arms" Environments, using several sizes and trained with these topologies. Table 1 presents the training results which shows clearly that the DDPG size does not need to be big in size, in the contrary the small size networks (#3, #4, and #5) shows significantly faster and better performance than the big ones (#2). All the networks are trained with the hyper parameters listed in Table 2.

Table 1 Training Results for Different DDPG & D4PG Topologies.

#	Version	Size	Episodes to reach 30	Training Time	Comment
1	1-Arm DDPG	Actor: Input Layer: 33 1st Hidden Layer: 128 2nd Hidden Layer: 64 Output Layer: 4 Critic: Input Layer: 33 1st Hidden Layer: 128 2nd Hidden Layer: 64 Output Layer: 1	2493	981.9 min	With CPU
2	1-Arm DDPG	Actor: Input Layer: 33 1st Hidden Layer: 256 2nd Hidden Layer: 128 Output Layer: 4 Critic: Input Layer: 33 1st Hidden Layer: 256 2nd Hidden Layer: 128 Output Layer: 1	3006	907.8 min	With CPU
3	1-Arm DDPG	Actor: Input Layer: 33 1st Hidden Layer: 128 2nd Hidden Layer: 64	1290	610.8 min	With CPU

		Output Layer: 4 Critic: Input Layer: 33 1st Hidden Layer: 64 2nd Hidden Layer: 32 Output Layer: 1			
4	1-Arm DDPG	Actor: Input Layer: 33 1st Hidden Layer: 128 2nd Hidden Layer: 64 Output Layer: 4 Critic: Input Layer: 33 1st Hidden Layer: 96 2nd Hidden Layer: 48 Output Layer: 1	4470	902.7 min	With CPU
5	20-Arms D4PG	Actor: Input Layer: 33 1st Hidden Layer: 128 2nd Hidden Layer: 64 Output Layer: 4 Critic: Input Layer: 33 1st Hidden Layer: 96 2nd Hidden Layer: 48 Output Layer: 1	181	64.5 min	With CPU
6	20-Arms D4PG	Actor: Input Layer: 33 1st Hidden Layer: 450 2nd Hidden Layer: 450 Output Layer: 4 Critic: Input Layer: 33 1st Hidden Layer: 250 2nd Hidden Layer: 250 Output Layer: 1	133	81.2 min	With CPU
7	20-Arms D4PG	Actor: Input Layer: 33 1st Hidden Layer: 350 2nd Hidden Layer: 350 Output Layer: 4 Critic: Input Layer: 33 1st Hidden Layer: 250 2nd Hidden Layer: 250 Output Layer: 1	136	43.3 min	With GPU
8	20-Arms D4PG	Actor: Input Layer: 33	127	41.3 min	With GPU

		1st Hidden Layer: 400 2nd Hidden Layer: 300 Output Layer: 4 Critic: Input Layer: 33 1st Hidden Layer: 400 2nd Hidden Layer: 300 Output Layer: 1			

Table 2 Hyper-parameters Values.

Hyper parameter	Value
Replay Buffer Size	100,000
Batch Size	128
Discount Factor (γ)	0.97
Soft-update for Target Parameters (τ)	0.001
Learning Rate (Actor)	0.0001
Learning Rate (Critic)	0.001
Weight Decay	0.000
Update Every (only for 1-Arm)	12 Samples

Figure 2 and Figure 3 depict the training performance of the DDPG & D4PG for 1-Arm and 20-Arms environments respectively to reach the goal of 30.

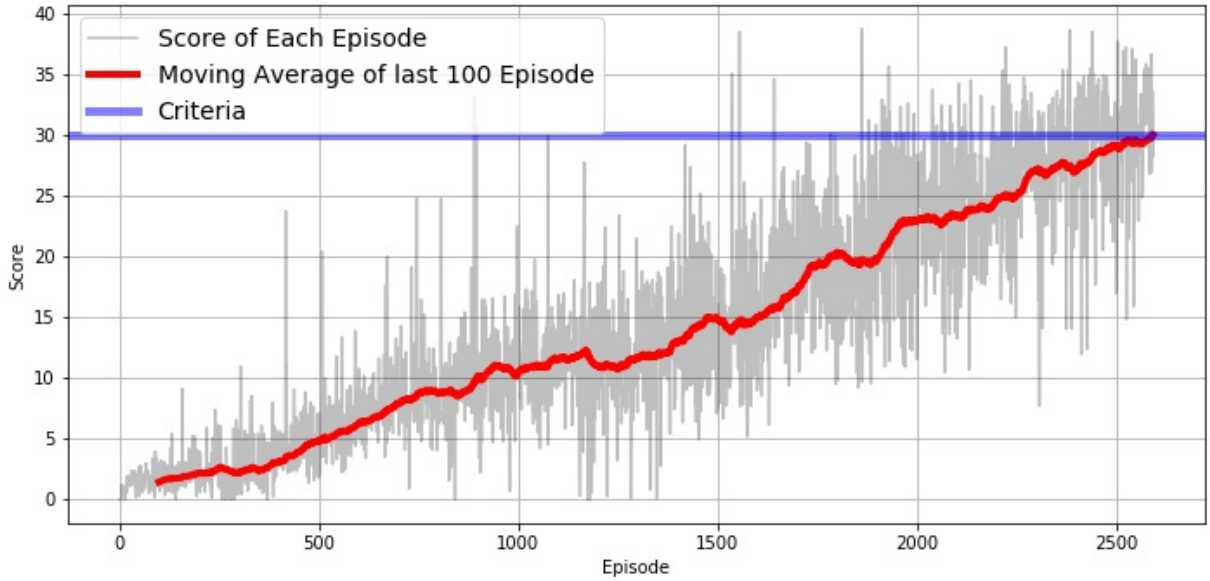

Figure 3 Training Performance of the DDPG(#1) to reach a 30.0 score.

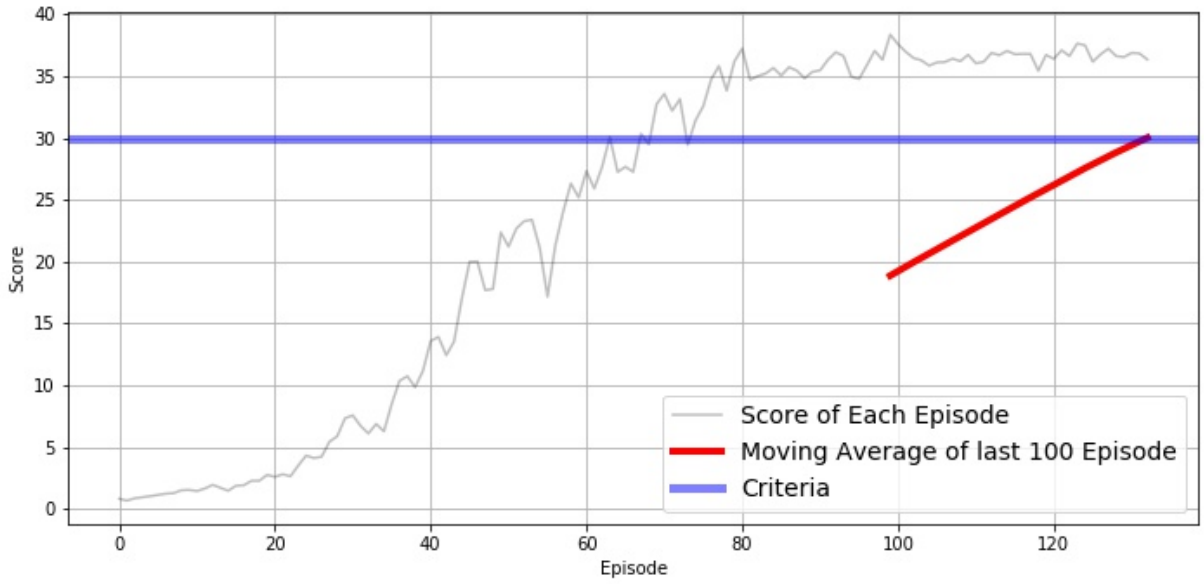

Figure 4 Training Performance of the D4PG(#6) to reach a 30.0 score.

4. Evaluation and Testing

After training, both the DDPG and D4PG algorithms are tested on 1-Arm and 20-Arms environments respectively using a different seed (2). The network was already trained with seed

(0). Figure 3 and Figure 4 depict the testing results, showing that some of the episodes collect a score of “38.59” and “37.1” respectively.

In [53]:
```
agent = Agent(state_size=state_size, action_size=action_size, random_seed=2)

checkpoint_actor_file  = 'checkpoint_actor.pth'
checkpoint_critic_file = 'checkpoint_critic.pth'

agent.actor_local.load_state_dict(torch.load(checkpoint_actor_file))
agent.critic_local.load_state_dict(torch.load(checkpoint_critic_file))

env_info = env.reset(train_mode=False)[brain_name]     # reset the environment
state    = env_info.vector_observations                # get the current state (for each agent)
scores   = np.zeros(num_agents)                        # initialize the score (for each agent)

for t in range(1000):
    action = agent.act(state, add_noise=False)

    env_info = env.step(action)[brain_name]
    next_state = env_info.vector_observations
    rewards = env_info.rewards
    dones = env_info.local_done

    state = next_state
    scores += rewards

    if np.any(dones):                                  # exit loop if episode finished
        break

print('Total score for the Agent in this episode: {}'.format(np.mean(scores)))
```

```
Total score for the Agent in this episode: 38.589999137446284
```

Figure 5 Testing Performance of the DDPG(#1) for an Individual Random Episode.

```
In [21]: agent = Agents(state_size=state_size, action_size=action_size, num_agents=num_agents, random_seed=2)

checkpoint_actor_file  = 'checkpoint_actor_v0.4.pth'
checkpoint_critic_file = 'checkpoint_critic_v0.4.pth'

agent.actor_local.load_state_dict(torch.load(checkpoint_actor_file))
agent.critic_local.load_state_dict(torch.load(checkpoint_critic_file))

env_info = env.reset(train_mode=False)[brain_name]     # reset the environment
state    = env_info.vector_observations               # get the current state (for each agent)
#print(state)
agent.reset()
scores   = np.zeros(num_agents)                        # initialize the score (for each agent)

for t in range(1000):
    action = agent.act(state)

    env_info = env.step(action)[brain_name]
    next_state = env_info.vector_observations
    rewards = env_info.rewards
    dones = env_info.local_done

    state = next_state
    scores += rewards

    if np.any(dones):                                  # exit loop if episode finished
        break

print('Total score for the Agent in this episode: {}'.format(np.mean(scores)))

Total score for the Agent in this episode: 37.10149917071685
```

Figure 6 Testing Performance of the D4PG(#6) for an Individual Random Episode.

4. Conclusion

The following are some concluding remarks that are deduced from the work of this project:

1. Both DDPG and D4PG as well as 1-Arm and 20-Arms environment are able to reach good solutions.
2. D4PG and the 20-Arms environment are superior to the DDPG and the 1-Arm environment in terms of the number of training episodes and the training time.
3. The architecture of both the Actor and Critic networks has a major effect on mainly the training convergence but not much the speed.
4. It is noticed that the size of the Actor must be bigger or at least the same size of the Critic for better convergence.
5. The learning rate of the Actor is recommended to be equal or less than the Critic learning rate.
6. Using the GPU helps to speed up the training speed by almost the double.

5. Future Improvement

There are several suggestions that can be carried out to further understand and improve the performance of the agent:

1. Trying other algorithms like REINFORCE, PPO, A3C, A2C, GAE, TRPO, and TNPG.
2. Test the replay buffer — Implement a way to enable/disable the replay buffer to measure the impact the replay buffer on performance.
3. Add prioritized experience replay — Rather than selecting experience tuples randomly, prioritized replay selects experiences based on a priority value that is correlated with the magnitude of error. This can improve learning by increasing the probability that rare and important experience vectors are sampled.
4. Studying more the effect the noise parameters of the *Ornstein-Uhlenbeck Process* on the training performance and the convergence.

References

[1] Unity's Reacher environment, https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Examples.md#reacher, retrieved on Oct. 19th, 2019.

[2] Richard S. Satun, Andew G. Barto, "Rienforcement Learning – An Introducation", 2nd Edition, The MIT Press, 2018.

[3] Andrej Karpathy, "Deep Reinforcement Learning: Pong from Pixels", http://karpathy.github.io/2016/05/31/rl/, May 31, 2016, retrieved on Oct. 19th, 2019.

[4] Open AI, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning", https://openai.com/blog/evolution-strategies/, retrieved on Oct. 19th, 2019.

[5] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov, "Proximal Policy Optimization Algorithms", arXiv:1707.06347v2 [cs.LG] 28 Aug 2017.

[6] Shixiang Gu, Timothy Lillicrap, Zoubin Ghahramani, Richard E. Turner, Sergey Levine, "Q-PROP: Sample-Efficient Policy Gradient With an Off-Policy Critic", arXiv:1611.02247v3 [cs.LG] 27 Feb 2017.

[7] Open AI, "Open AI Baselines: ACKTR & A2C", https://openai.com/blog/baselines-acktr-a2c/, retrieved on Oct. 19th, 2019.

[8] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan and Pieter Abbeel, "High-Dimentional Continous Control Using Generalized Advantage Estimation", arXiv:1506.02438v6 [cs.LG] 20 Oct 2018.

[9] Timothy P. Lillicrap , Jonathan J. Hunt , Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver & Daan Wierstra, "Continous Control With Deep Reinforcement Learning", arXiv:1509.02971v6 [cs.LG] 5 Jul 2019.

[10] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine, "Continuous Deep Q-Learning with Model-based Acceleration", arXiv:1603.00748v1 [cs.LG] 2 Mar 2016.

[11] Gabriel Barth-Maron, Matthew W. Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva TB, Alistair Muldal, Nicolas Heess, Timothy Lillicrap, "Distributed Distributional Deterministic Policy Gradients", arXiv:1804.08617v1 [cs.LG] 23 Apr 2018.

[12] Chris Yoon, "Deep Deterministic Policy Gradients Explained", https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b, retrieved on Oct. 19th, 2019.

[13] Ruishan Liu, James Zou, "The Effects of Memory Replay in Reinforcement Learning", arXiv:1710.06574v1 [cs.AI] 18 Oct 2017.

[14] Wikipedia, "Ornstein–Uhlenbeck process", https://en.wikipedia.org/wiki/Ornstein%E2%80%93Uhlenbeck_process, retrieved on Oct. 19th, 2019.

[15] Edouard Leurent, "Why do we use the Ornstein Uhlenbeck Process in the exploration of DDPG?", https://www.quora.com/Why-do-we-use-the-Ornstein-Uhlenbeck-Process-in-the-exploration-of-DDPG/answer/Edouard-Leurent?ch=10&share=4b79f94f&srid=udNQP