Multi-Agent Tennis Playing using Deep Distributed Distributional Deterministic Policy Gradients (D4PG)

By

Wael Farag

1. Goal

In this project, the goal is to train two agents who are manipulating two tennis rackets in order to bounce a ball over a net between them back and forth for as long as possible without dropping the ball or push outside the field bounds as shown in Figure 1. Each of the two agents has access to the environment's 8-dimensional state, which consists of the position and velocity of both the ball and the rackets. Each of these 8-dimensional states receives continuous control commands (signals) from each racket agent. Throughout the training, each agent learns which of the four actions it should take in each scenario. Each agent receives a reward of +0.1 if the agent hit the ball over the net successfully. If the agent lets the ball hit the ground or bounces it out of bounds, it receives a reward (penalty) of -0.01. Each agent has two continuous actions available, one for moving towards or away from the net, and one for jumping. The environment is considered solved if both the agents together accumulated an average a score of +0.5 over 100 consecutive episodes (where the maximum is taken over the two agents).

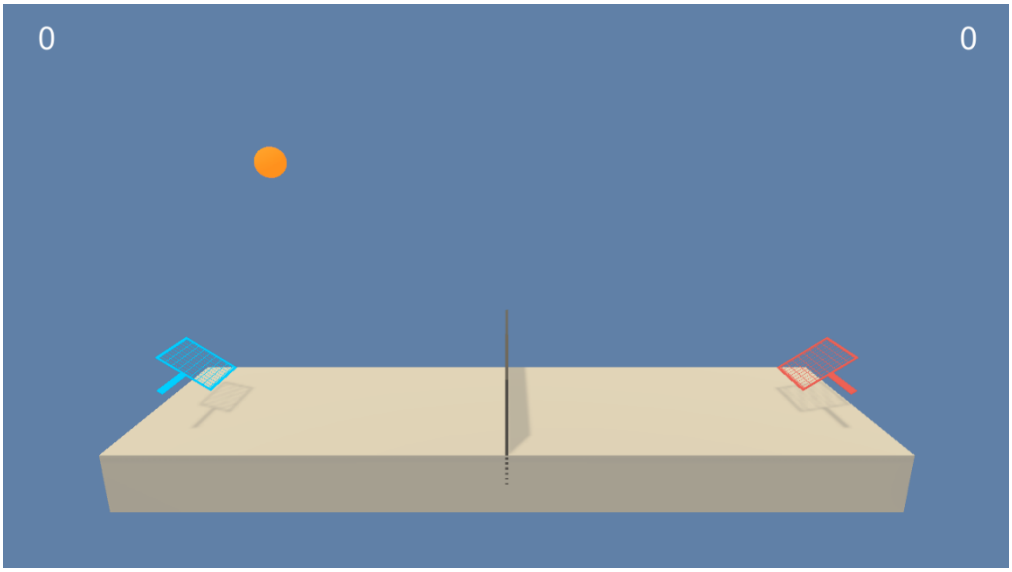

Figure 1 Multiple-Agent Tennis Environment.

The main goal of this project is to design and implement the two Agents to control the movement the tennis rackets. 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 move (step) that the agent was successfully to receive the ball correctly and to send it over the net. Thus, the goal of the agent is to keep receiving and sending back the ball for as many time steps as possible.

The observation space consists of 8 variables corresponding to position and velocity of both the ball and the rackets. Each action is a vector with two numbers, corresponding to the movement of the racket towards or away from the net. 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 agents must collectively receive rewards and aggregate them for the latest 100 consecutive episodes. The environment is considered solved, when the average (over 100 episodes) of the agents collective score is at least +0.5.

2. Approach

In this project, the Deep Distributed Distributional Deterministic Policy Gradients (D4PG) algorithm is used [11]. The algorithm is adopted to be used in the multi-agent environment. The tennis environment is built by unity technologies [1] and can be downloaded from this link. This environment will be solved using two agents trained using a tailored D4PG algorithm [11]. The training task is episodic, and in order to solve this environment, the agent must get an average score of +0.5 over 100 consecutive episodes.

The approach taken in this work starts from the DDPG algorithm [9-10] that is shown in Figure 2, and includes a number of enhancements. These extensions include a distributional critic update, the use of distributed parallel actors, N-step returns, and prioritization of the experience replay. The whole algorithm can be described by the pseudo code in Figure 3. For the matter of completeness, the multi-agent DDPG pseudo code is provided below:

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.

Algorithm 1 D4PG

Input: batch size M, trajectory length N, number of actors K, replay size R, exploration constant ϵ, initial learning rates α_0 and β_0

Initialize network weights (θ, w) at random
Initialize target weights $(\theta', w') \leftarrow (\theta, w)$
Launch K actors and replicate network weights (θ, w) to each actor
for $t = 1, \ldots, T$ **do**
  Sample M transitions $(\mathbf{x}_{i:i+N}, \mathbf{a}_{i:i+N-1}, r_{i:i+N-1})$ of length N from replay with priority p_i
  Construct the target distributions $Y_i = \left(\sum_{n=0}^{N-1} \gamma^n r_{i+n}\right) + \gamma^N Z_{w'}(\mathbf{x}_{i+N}, \pi_{\theta'}(\mathbf{x}_{i+N}))$
  Note, although not denoted the target Y_i may be projected (e.g. for Categorical value distributions).
  Compute the actor and critic updates

$$\delta_w = \frac{1}{M} \sum_i \nabla_w (R p_i)^{-1} d(Y_i, Z_w(\mathbf{x}_i, \mathbf{a}_i))$$

$$\delta_\theta = \tfrac{1}{M} \sum_i \nabla_\theta \pi_\theta(\mathbf{x}_i)\ \mathbb{E}[\nabla_{\mathbf{a}} Z_w(\mathbf{x}_i, \mathbf{a})]\big|_{\mathbf{a}=\pi_\theta(\mathbf{x}_i)}$$

  Update network parameters $\theta \leftarrow \theta + \alpha_t\, \delta_\theta$, $w \leftarrow w + \beta_t\, \delta_w$
  If $t = 0 \mod t_{\text{target}}$, update the target networks $(\theta', w') \leftarrow (\theta, w)$
  If $t = 0 \mod t_{\text{actors}}$, replicate network weights to the actors
end for
return policy parameters θ

Actor

repeat
  Sample action $\mathbf{a} = \pi_\theta(\mathbf{x}) + \epsilon \mathcal{N}(0, 1)$
  Execute action $\mathbf{a}$, observe reward r and state $\mathbf{x}'$
  Store $(\mathbf{x}, \mathbf{a}, r, \mathbf{x}')$ in replay
until learner finishes

Figure 3 Pseudo Code for D4PG algorithm.

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 $y^j = r_i^j + \gamma\, Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      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 4 Pseudo Code for Multi-Agent DDPG algorithm.

The Multi-Agent D4PG algorithm is basically constructed by using the Multi-Agent DDPG algorithm shown in Figure 4, and replacing the DDPG algorithm with the D4PG algorithm shown in Figure 3.

4. Experimentations and Results

The Deep Distributed Distributional Deterministic Policy Gradients (D4PG) algorithm is implemented using several actor and critic sizes and trained with the resultant topologies. Table 1 presents the finally tuned hyper-parameters used the training. Moreover, Table 2 details the internal structure of the Actor network, and Table 3 details as well the internal structure of the Critic network. Each agent consists of one actor networks plus one critic network.

Table 1 Hyper-parameters Values.

Hyper parameter	Value
Replay Buffer Size	1000,000
Batch Size	64
Discount Factor (γ)	0.97
Soft-update for Target Parameters (τ)	0.001
Learning Rate (Actor)	0.0005
Learning Rate (Critic)	0.005
Weight Decay	0.000
Update Every	2 Samples
No. of Steps	5
V_{max}, V_{min}	0.7, -0.7
No. of Atoms	51

Table 2 Actor Network Structure for One Agent.

Layer	Dimension
Input	24
1st Hidden: Linear Layer, Leaky Relu	256
2nd Hidden: Linear Layer, Leaky Relu	128
3rd Hidden: Linear Layer, Leaky Relu	2
Batch normalization 1D	2
Tanh Output	2

Table 3 Critic Network Structure for One Agent.

Layer	Dimension
Input	24
1st Hidden: Linear Layer, Leaky Relu	256
2nd Hidden: Linear Layer + Actor Output, Leaky Relu	256+2
3rd Hidden: Linear Layer, Leaky Relu	128
Output: Linear Layer	51

Figure 5 depict the training performance of the Multi-Agent D4PG to reach the ambitious goal of accumulating rewards of "**1.5**" score instead of the required score of "0.5". Moreover, Figure 6 shows the performance of each individual agent, which indicates clearly how close they performed.

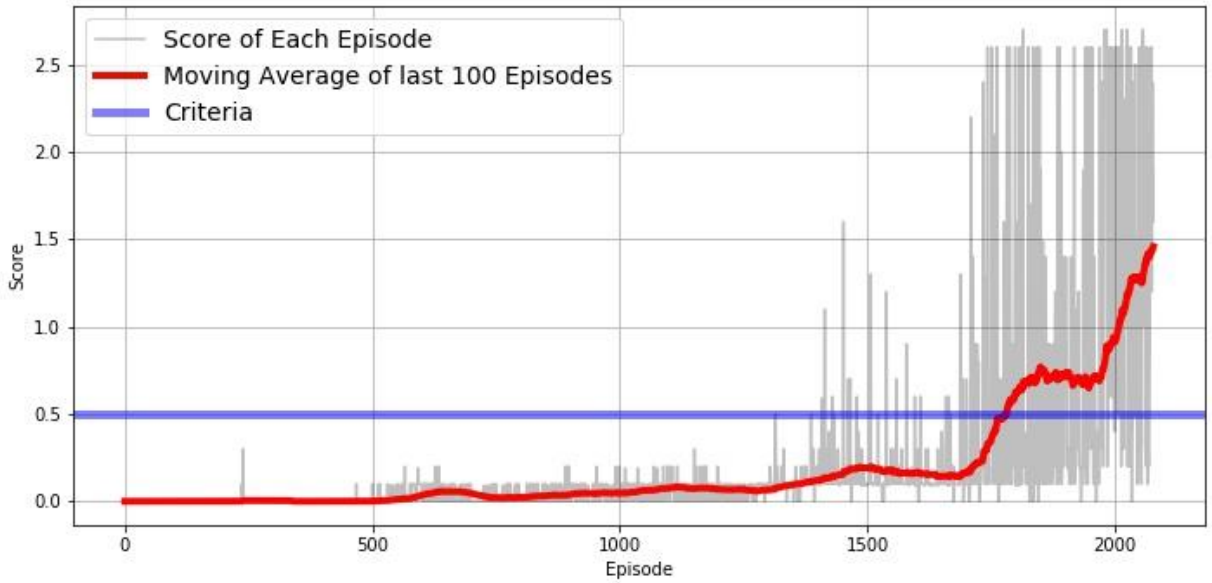

Figure 5 Training Performance of the D4PG to reach a 1.5 score.

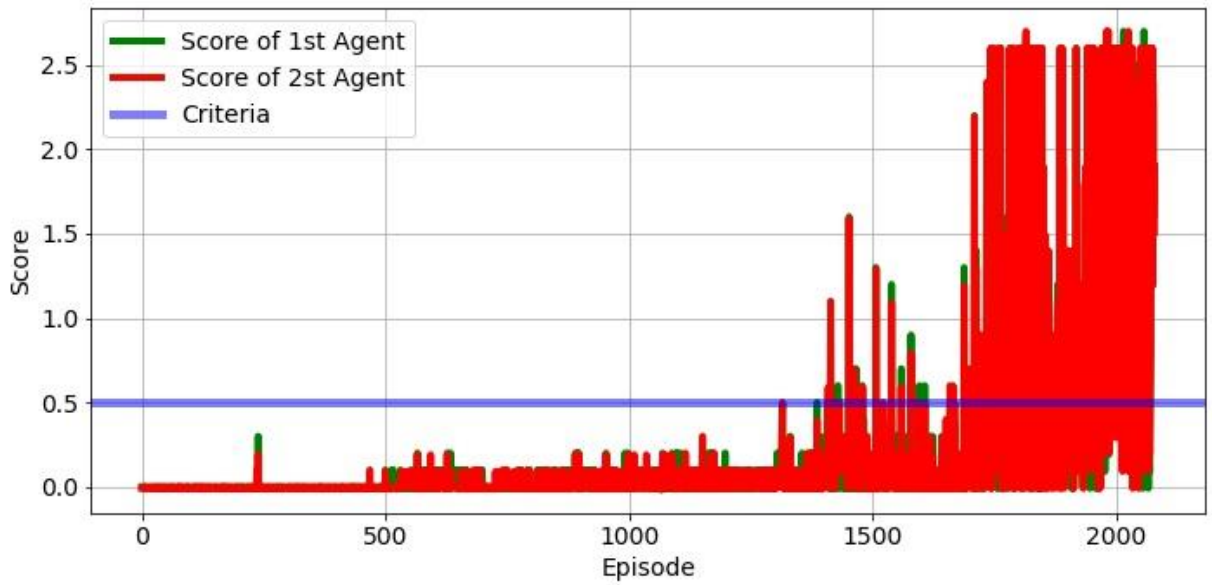

Figure 6 Training Performance of each individual D4PG agent.

4. Evaluation and Testing

After training, both the D4PG algorithms is tested on the tennis environment for 3 consecutive episodes using a different seed “42” other than the training seed which was “1”. Figure 7 and Figure 8 depict the testing results, showing that some of the episodes collect impressive scores of “2.7” and “2.6”. However, some episodes fails to reach a score od “0.1”.

```
#Watch trained Agents by multi-agent d4pg
from unityagents import UnityEnvironment

env = UnityEnvironment(file_name="Tennis_Windows_x86_64/Tennis.exe")
# get the default brain
brain_name = env.brain_names[0]
brain = env.brains[brain_name]

env_info = env.reset(train_mode=False)[brain_name]
num_agents = len(env_info.agents)
action_size = brain.vector_action_space_size
states = env_info.vector_observations
state_size = states.shape[1]


Agent1_Actor_checkpoint_latest_file  = "Agent1_Actor_checkpoint_latest.pth"
Agent2_Actor_checkpoint_latest_file  = "Agent2_Actor_checkpoint_latest.pth"

multiagent = multiAgent(state_size=state_size, action_size=action_size, seed=42,
                        BUFFER_SIZE=int(1e6), BATCH_SIZE=64, GAMMA=0.97, TAU=1e-3,
                        LR_ACTOR=5e-4, LR_CRITIC=5e-4, WEIGHT_DECAY=0.0, UPDATE_EVERY=2, N_step=5)

multiagent.d4pg_Agents[0].actor_local.load_state_dict(torch.load(Agent1_Actor_checkpoint_latest_file), strict=False)
multiagent.d4pg_Agents[1].actor_local.load_state_dict(torch.load(Agent2_Actor_checkpoint_latest_file), strict=False)


for i in range(1, 4):                                      # play game for 3 episodes
    env_info = env.reset(train_mode=False)[brain_name]     # reset the environment
    states = env_info.vector_observations                  # get the current state (for each agent)
    scores = np.zeros(2)                                   # initialize the score (for each agent)
    while True:
        actions = multiagent.acts(states)                  # select an action (for each agent)
        env_info = env.step(actions)[brain_name]           # send all actions to tne environment
        next_states = env_info.vector_observations         # get next state (for each agent)
        rewards = env_info.rewards                         # get reward (for each agent)
        dones = env_info.local_done                        # see if episode finished
        scores += env_info.rewards                         # update the score (for each agent)
        states = next_states                               # roll over states to next time step
        if np.any(dones):                                  # exit loop if episode finished
            break
    print('Score (max over agents) from episode {}: {}'.format(i, np.max(scores)))

env.close()
```

```
Score (max over agents) from episode 1: 0.7000000104308128
Score (max over agents) from episode 2: 0.9000000134110451
Score (max over agents) from episode 3: 1.2000000178813934
```

Figure 7 Testing Performance of the D4PG for 3 consecutive random episodes.

```
Score (max over agents) from episode 1: 2.600000038743019
Score (max over agents) from episode 2: 2.7000000402331352
Score (max over agents) from episode 3: 0.10000000149011612
```

Figure 8 Testing Performance of the D4PG for another set of 3 consecutive episodes.

4. Conclusion

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

1. The D4PG algorithm performs very well to train the agents for the tennis environment, and is able to reach good solutions.
2. The architecture of both the Actor and Critic networks has a major effect on mainly the training convergence but not much on the speed.
3. It is noticed that the size of the Actor must be bigger or at least the same size of the Critic for better convergence.
4. The learning rate of the Actor is recommended to be equal or less than the Critic learning rate. Several learning rates are tested and learning rate of "1e-3" was large so that convergence was not reached easily and "1e-4" was very slow. Therefore. "5e-4" was a good compromise.
5. Using the GPU helps to speed up the training speed by almost 50%. The training that takes 57.6 minutes on CPU, takes 41.3 minutes on GPU.

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. Implementing Rainbow Algorithm [16] which combines good features from different algorithms to form an integrated agent.
3. Test the replay buffer — Implement a way to enable/disable the replay buffer to measure the impact the replay buffer on performance.
4. In this implementation, the prioritized experience replay is used. Testing the effect of the prioritized experience replay is recommended. Rather than selecting experience tuples randomly, prioritized replay selects experiences based on a priority value that is correlated with the magnitude of error. This is believed to improve learning by increasing the probability that rare and important experience vectors are sampled.
5. 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

[16] Matteo Hesse et al, "Rainbow: Combining Improvements in Deep Reinforcement Learning", arXiv:1710.02298v1 [cs.AI] 6 Oct 2017.