Robot-Arm Navigation using Deep Deterministic Policy Gradient Algorithms

Wael Farag

College of Eng. & Tech., American University of the Middle East, Kuwait.
Electrical Eng., Cairo University, Egypt
wael.farag@aum.edu.kw, wael.farag@cu.edu.eg

***Abstract*— In this paper, the Deep Deterministic Policy Gradient (DDPG) reinforcement learning algorithm is employed to enable a double-jointed robot arm to reach continuously-changing target locations. The experimentation of the algorithm is carried out by training an agent to control the movement of this double-jointed robot arm. The architectures of the actor and cretic networks are meticulously designed and the DDPG hyperparameters are carefully tuned. An enhanced version of the DDPG is also presented to handle multiple robot arms simultaneously. The trained agents are successfully tested in the Unity Machine Learning Agents environment for controlling both a single robot arm as well as multiple simultaneous robot arms. The testing shows the robust performance of the DDPG algorithm for empowering robot arm maneuvering in complex environments.**

Keywords—Reinforcement Learning, Policy-Gradients Methods, DDPG, Machine Learning, Robotics, Robot Arm.

I. INTRODUCTION

The development of the artificial intelligence field in the real-world is measured by the ability to solve complex control tasks [1][2] that possess a high dimension in input and action spaces [3][4]. Reinforcement Learning (RL) lends itself as a high-potential player in this field since the advent of the Deep Q Network (DQN) [5], as it achieved human-level performance in several early-level 2-dimensional computer games. Moreover, reinforcement learning has been developed further to beat human-level performance in board games like Go [6] and Chess [7]. However, the mentioned games and the applied algorithms still have a finite number (although sometimes huge) of discrete actions.

In most control tasks in real applications (like robotics and autonomous driving [8][9]), continuous action spaces are the most common. Therefore, if algorithms like DQN (which depends on the maximization of its policy value function for action selection) need to be applied to such control tasks, it needs to go through a computationally expensive optimization step or a careful discretization step of the action space. The most common approach, in this case, is discretization, however, it shows on several occasions that it is an insufficient approximation particularly in high-dimensional configurations or in delicate cases that requires very fine and precise control actions. Consequently, a more logical approach is exploited that depends on the explicit parameterization of a certain policy and optimizing its long-term value while following this policy. The methods that follow this approach are referred to as Policy-Based Methods (PBM) [10].

In this work, the focus will be on the PBMs and its subcategory Policy Gradient Methods (PGM) [11]. Several research endeavors have been conducted to explore further and enhance PBMs and PGMs algorithms. Examples like REINFORCE Algorithm [12], Proximal Policy Optimization (PPO) algorithms [13], Asynchronous Advantage Actor-Critic (A3C) algorithms [14], Advantage Actor-Critic (A2C) algorithms [15], Generalized Advantage Estimation (GAE) algorithms [16], Trust Region Policy Optimization (TRPO) algorithm [17], Truncated Natural Policy Gradient (TNPG)

algorithm [17], Deep Deterministic Policy Gradient (DDPG) algorithms [17][18], and Distributed Distributional Deep Deterministic Policy Gradients (D4PG) [19] which integrates several modifications to the DDPG algorithm.

To be more specific, the DDPG reinforcement learning algorithm [17] is mainly considered in this work. The fundamental operation of DDPG is based on an off-policy actor-critic principle, in which the update of the actor-network particularly depends on a learned critic only. Therefore, any enhancements to the critic learning process will positively affect the updates of the actor. DDPG is a model-free approach that can learn competitive policies using low-dimensional observations (e.g. Cartesian coordinates or joint angles of a multi-joint robot arm).

The DDPG algorithm is implemented using deep learning to realize a deterministic policy, in which the Actor-Critic framework is employed. The output of the policy is not the probability of action but a certain action. As a result, the sampled data is reduced and DDPG can guarantee the continuity and the smoothness of the joint movement.

The main purpose of the work in this paper is to adopt the DDPG algorithm to solve control problems that allow the successful manipulation of a robot arm with multiple joints, and multiple degrees of freedom to reach and grasp a ball that moves randomly in a 3D space as shown in Fig. 1.

The process of finding the most appropriate actor and cretic topologies will be presented. The final internal structures of the actor and critic networks will be described in detail, and the tuning of the hyperparameters will be also explained. The training results are demonstrated with discussion. Testing, evaluation, concluding remarks and future work will be presented as well.

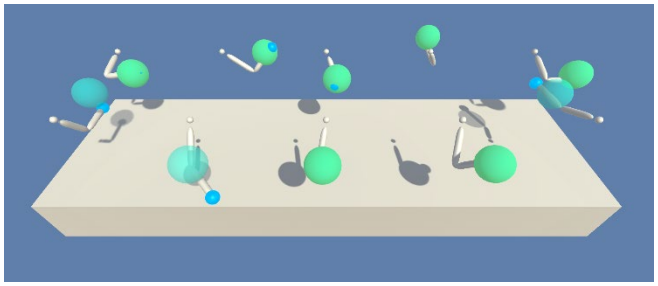

Fig. 1. The Multiple-Agent Tennis Environment.

II. Overview of The DDPG Algorithm

In the typical reinforcement learning setup, an agent interacts with a certain environment in discrete time. At each time step t, the agent makes observations $x_t \in \mathrm{X}$ takes action $a_t \in \mathrm{A}$, and receives reward $r(x_t, a_t) \in \mathbb{R}$. The setup environment has a real action space $\mathrm{A} \in \mathbb{R}^d$.

The control of the agent behavior is carried out using a policy $\pi: \mathrm{X} \rightarrow \mathrm{A}$ that maps each observation to action. The expected return is described using the state-action value function $Q_\pi(a, x)$, under the condition of first taking an arbitrary action $a \in \mathrm{A}$ from a certain state $x \in \mathrm{X}$ and subsequently acting according to π. Hence, the value function is defined as

$$Q_\pi(a, x) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(x_t, a_t)\right] \tag{1}$$

where $x_0 = x, \;\; a_0 = a, \;\; a_t = \pi(x_t), x_t \sim p(.|x_{t-1}, a_{t-1})$ is the transition kernel (probability) to next state, and $\gamma \in [0,1]$ is the discount factor. The quality of a policy is commonly evaluated using Eq. (1). Even though it is possible to derive an updated policy from Q_π. However, such an approach typically requires maximizing the value function with respect to

a and is made complicated by the existence of continuous action space. Instead, a parametrized policy π_θ is formed and the expected value of this policy is maximized by optimizing the objective function $J(\theta) = \mathbb{E}\left[Q_{\pi_\theta}\left(x, \pi_\theta(x)\right)\right]$. By incorporating the deterministic policy gradient theorem [19], the gradient of this objective function is composed as

$$\nabla_\theta J(\theta) \approx \mathbb{E}_\rho\left[\nabla_\theta \pi_\theta(x) \nabla_a Q_{\pi_\theta}(x, a)|_{a=\pi_\theta(x)}\right] \tag{2}$$

where ρ is the state-visitation distribution associated with some behavior policy. The behavior policy is different than the policy π, consequently, it is possible to evaluate the behavior policy ρ from data gathered off-policy. The gradient given by Eq. (2) can be approximated and modeled using a parameterized critic $Q_w(x,\, a)$, as the exact gradient assumes access to the true value of the current policy which is unreachable.

By introducing the Bell operator

$$(\mathrm{T}_\pi Q)(x, a) = r(x, a) + \gamma \mathbb{E}\left[Q\left(x', \pi(x')\right)|x,\, a\right] \tag{3}$$

which include an expectation value calculated with respect to the next state x', the Temporal Difference (TD) error can be minimized. TD is the difference between the value function before and after the application of the Bellman update. Usually, the TD error is evaluated under separate target policy and value networks, (i.e. networks with separate parameters (θ', w')), to stabilize the learning process. By taking the two-norm of this error we can write the resulting loss as

$$L(w) = \mathbb{E}_\rho\left[\left(Q_w(x, a) - \left(T_{\pi_\theta}, Q_{w'}\right)(x, a)\right)^2\right] \tag{4}$$

In practice, the target networks will be periodically replaced with copies of the current network weights. Ultimately, by training a deep neural network that represents the policy using the deterministic policy gradient in Eq. (2) and training another deep neural to minimize the TD error in Eq. (4), the Deep Deterministic Policy Gradient (DDPG) algorithm [17] is obtained as illustrated in Fig. 2. The actor produces an action given the current state of the environment, and the critic produces a TD error signal given the state and resultant reward. For the critic to estimate the action-value function, it needs to acquire the output of the actor as one of its inputs. Therefore, the critic uses the next-state value (TD target) that is generated from the current action by the given environment. The output of the critic drives the learning process in both the actor and the critic.

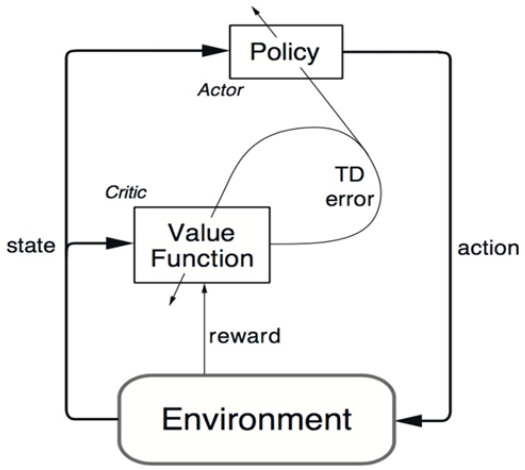

Fig. 2. The workflow of the DDPG algorithm.

To further enhance the DDPG algorithm, the sample-based approximation to its gradients is preferably employed by using data gathered in some replay tables as it shows better convergence.

Next, a modification to the DDPG update is considered in which it utilizes *N*-step returns when estimating the TD error. This can be seen as replacing the *Bellman operator* T_π^N with an *N*-step variant as follows:

$$(\mathrm{T}_\pi^N Q)(x_o, a_o) = r(x_o, a_o) + \mathbb{E}\left[\sum_{n=1}^{N-1} \gamma^n r(x_n, a_n) + \gamma^N Q\big(x_N, \pi(x_N)\big) | x_o, a_o\right] \tag{8}$$

where the expectation is with respect to the *N*-step transition dynamics. The process of gathering experience is distributed by modifying the standard training procedure. As per Eq. (2) and (4), the actor and critic updates rely on sampling from some state-visitation with distribution ρ. Nevertheless, as a significant enhancement included in the D4PG algorithm [19], the process of gathering experience can be parallelized by writing to the same reply table in parallel by using K independent actors. Moreover, during the learning process, the training algorithm can then sample from the previously constructed replay table of size R and perform the necessary network updates using this data. Furthermore, the sampling can be carried out using non-uniform priorities p_i as in [20]. However, this requires the use of what is called "importance sampling", which is implemented by weighting the critic update by a factor of $1/R_{p_i}$.

The actor (the policy function) and critic (the value function) parameters are updated using stochastic gradient descent with learning rates, α_i and β_i respectively, which are adjusted online using ADAM [21] for T iterations in the episode.

III. The Setup of The Environment

The simulation environment for training, testing, and evaluating the DDPG RL algorithm is built using Unity Technologies [22]. Two versions of the environment have been deployed:

1. The 1st version includes only a single double-jointed robot arm. This environment should be solved using a single agent trained using a designed Deep Deterministic Policy Gradient (DDPG) algorithm [16]. The training task is episodic, and to solve this environment, the agent must get at least an average score of +30 over 100 consecutive episodes.
2. The 2nd version is the 20 double-jointed arms similar to the one shown in Fig. 1. This environment will be solved using 20 agents trained simultaneously using the designed Distributed Distributional Deep Deterministic Policy Gradients (D4PG) [19] as it uses multiple (noninteracting, parallel) copies of the same agent to distribute the task of gathering experience. The barrier for solving this environment is slightly different, to take into account the presence of many agents (20 in this case). In particular, the agents must get an average score of +30 (over 100 consecutive episodes, and including all the agents). After each episode, and to get a score for each agent, the rewards are added up for each agent separately (without discounting). 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 including all the 20 agents).

The observation space consists of 33 variables corresponding to the 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 experiment 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 planes, with multiple Z values).

The action space is a vector with four numbers, clamped between -1 and 1, corresponding to the X and Z torques applicable to the two joints of the arm ("shoulder" and "elbow"). Fig. 1 shows a version of the environment with 10 arms follow the goal spheres movements.

A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of your agent is to maintain its position at the target location for as many time steps as possible.

IV. Experimentations and Results

The DDPG algorithm is implemented for the “Single Arm” environment and the D4PG is implemented for the “20-Arms” environment, using several actor-critic topologies and sizes, and the training is performed separately for each version. After extensive trials, TABLE . presents samples of these training results which show clearly that the size of the DDPG and D4PG networks does not have to be big to achieve higher performance, on the contrary, the small-size networks (#3) show significantly faster and better performance than the big ones (#1, #2 and #4). Nevertheless, it is needed to mention here that all the networks are trained with the hyperparameters listed in TABLE .

Employing the GPU (NVIDIA Tesla K80, 13GB RAM) [23] instead of the CPU (Intel Xeon Processor @2.3GHz (1 core, 2 threads), 13GB RAM) has reduced the training time by almost 50%, as demonstrated by comparing cases (#7 and #8) with cases (#5 and #6).

During training, both the DDPG and D4PG algorithms employ the *Ornstein-Uhlenbeck Process* (OUP) [24] to improve the exploration phase. It generates temporally correlated noise centered around “0” to explore well in physical environments that have momentum. The OUP is used with $\theta = 0.15$ and $\sigma = 0.2$ as in [25].

TABLE I. Training Results for Different DDPG & D4PG Topologies.

#	Version	Size	#EP*	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	3006	907.8 min	With CPU

		1st Hidden Layer: 256 2nd Hidden Layer: 128 Output Layer: 1			
3	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: 64 2nd Hidden Layer: 32 Output Layer: 1	1290	610.8 min	With CPU
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	136	43.3 min	With GPU

		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			
8	20-Arms D4PG	Actor: Input Layer: 33 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	127	41.3 min	With GPU

*Number of Episodes to reach 30.

TABLE II. HYPER-PARAMETERS VALUES.

Hyperparameter	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

Fig. and Fig. depict the training performance of the DDPG & D4PG for 1-Arm and 20-Arms environments respectively to reach the goal of "a score of 30 for the 100-episode average". In both cases, the training advanced smoothly to reach the target score.

The D4PG algorithm shows impressive performance if compared to that of the DDPG. The training time of D4PG represents on average 8.6% of the training time of the DDPG, even though the number of iterations in both algorithms is almost the same (the average is 2815 for DDPG and 2885 for D4PG), noting that each D4PG iteration is equivalent to 20 iterations of the DDPG. As has been explained before, the main difference between the two algorithms is that the D4PG uses K parallel actors ($K=20$) learning simultaneously from data sampled from a shared experience reply buffer (distributed learning).

After the training phase is accomplished, both the DDPG and D4PG algorithms are tested on 1-Arm and 20-Arms environments respectively using a different seed (equals 2). The network was already trained with a seed (0). Fig. 5 and Fig. 6 present the code snippets and the testing results, showing that an episode of 1000 time steps scored "38.59" and "37.1" respectively. More extensive experimentation produces the same robust performance for both algorithms.

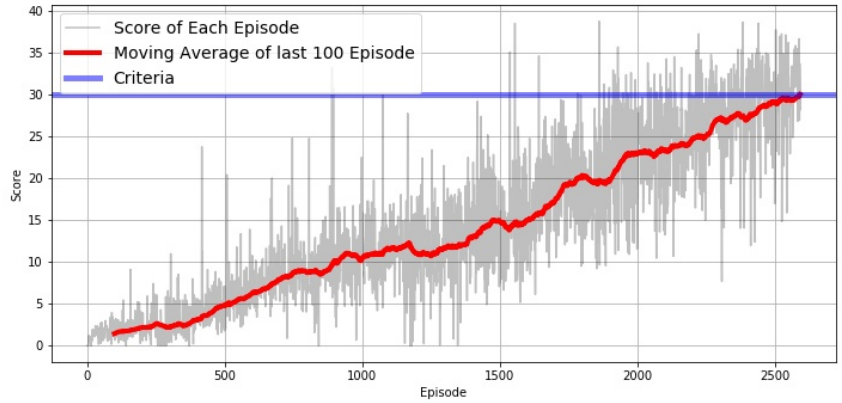

Fig. 3. Training Performance of the DDPG (#1) to reach a 30.0 score.

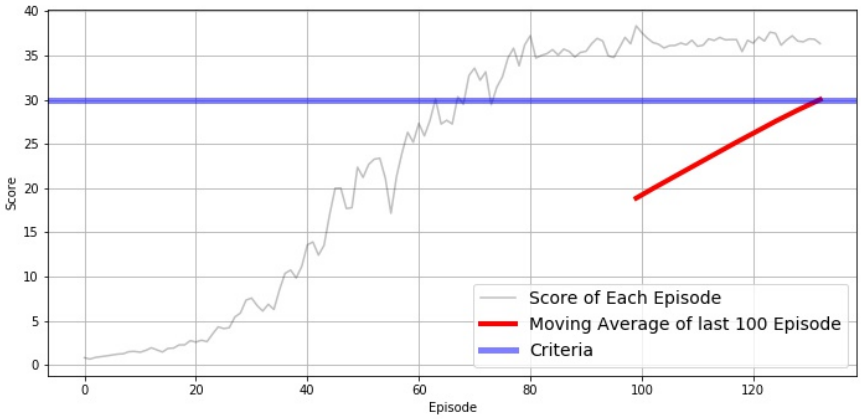

Fig. 4. Training Performance of the D4PG (#6) to reach a 30.0 score.

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
```

Fig. 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
```

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

V. CONCLUSION

The DDPG and D4PG reinforcement learning algorithms are used to train a double-jointed robotic arm to reach an object moving in a 3-D space around the arm. The DDPG is used to train a single arm while the D4PG is used to train multiple arms simultaneously. The D4PG is an improved version of the DDPG that uses multiple independent actors learning together in parallel and write together to a shared experience reply buffer. This improvement allows building models that command multiple robot arms instead of one. This improvement as well shows superior learning performance as it reduced the training time by almost 90% with a highly reduced number of training episodes. However, testing the trained models (agents) from both the DDPG and the D4PG algorithms shows very comparable robust performance.

The architecture of both the actor and critic networks of both the DDPG and D4PG show a significant impact on mainly the training convergence but not much on the convergence speed. It is observed that the size of the actor should be bigger or at least the same size as the critic for a better convergence, while the learning rate of the critic is recommended to be larger. Moreover, employing a GPU speed up the training process by almost double. Furthermore, both the DDPG and D4PG algorithms successfully employ the *Ornstein-Uhlenbeck Process* to improve the exploration phase.

Finally, the experimentation and the testing results show the robust performance of the DDPG algorithm for empowering robot arm maneuvering in complex environments.

VI. FUTURE WORK

Several suggestions can be carried out to further understand and improve the performance of the Reacher:

1. Trying other algorithms like REINFORCE, PPO, A3C, A2C, GAE, TRPO, and TNPG, and compare the results with the DDPG.
2. Applying the D4PG algorithm for a 3-joint robot arm environment.
3. Implementing Rainbow Algorithm [26] which combines good features from different algorithms to form an integrated agent.

4. Another avenue of future research is to use Multi-Agent DDPG (MADDPG) [27][28] to coordinate the work of multiple cooperating robot arms to fulfill a single task.
5. Studying more deeply the effect of the noise parameters of the Ornstein-Uhlenbeck Process [24] on the training performance and the convergence.

REFERENCES

[1] W. Farag, "Complex Trajectory Tracking Using PID Control for Autonomous Driving", *International Journal of Intelligent Transportation Systems Research*, Springer, Sept., (2019).

[2] W. Farag, "Complex-Track Following in Real-Time Using Model-Based Predictive Control", *International Journal of Intelligent Transportation Systems Research*, Springer, June, (2020).

[3] Wael Farag, "Synthesis of intelligent hybrid systems for modeling and control", Ph.D. Thesis, *University of Waterloo*, Canada, (1998).

[4] WA Farag, VH Quintana, G Lambert-Torres, "Neuro-Fuzzy Modeling of Complex Systems Using Genetic Algorithms", *IEEE International Conference on Neural Networks* (IEEE ICNN'97) 1, pp. 444-449, 1997.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A Rusu, J. Veness, M. G Bellemare, A. Graves, M. Riedmiller, A. K Fidjeland, G. Ostrovski, et al., "Human-level control through deep reinforcement learning", *Nature*, 518(7540):529–533, 2015.

[6] D. Silver, A. Huang, C. J Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. "Mastering the game of go with deep neural networks and tree search", *Nature*, 529(7587):484–489, 2016.

[7] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play", *Science,* Vol. 362, Issue 6419, pp. 1140-1144, Dec. 2018.

[8] Wael Farag, "Cloning Safe Driving Behavior for Self-Driving Cars using Convolutional Neural Networks", *Recent Patents on Computer Science*, Bentham Science Publishers, The Netherlands, Vol. 12, No. 2, pp. 120-127(8), (2019**)**.

[9] Wael Farag, Zakaria Saleh, "Road Lane-Lines Detection in Real-Time for Advanced Driving Assistance Systems", *Intern. Conf. on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT'18)*, Bahrain, 18-20 Nov., (2018).

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

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

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

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

[14] 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.

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

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

[17] 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.

[18] 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.

[19] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, Dhruva TB, A. Muldal, N. Heess, T. Lillicrap, "Distributed Distributional Deterministic Policy Gradients", arXiv:1804.08617v1 [cs.LG] 23 Apr 2018.

[20] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay", *Inter. Conf. on Learning Representations*, arXiv:1511.05952v4, 2016.

[21] D. Kingma and J. Ba, "Adam: A method for stochastic optimization", In Inter. Conf. on Learning Representations, San Diego, USA, 2015.

[22] Unity's Reacher environment, https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Examples.md#reacher, retrieved on Oct. 19th, 2020.

[23] M. Nagiub and W. Farag, "Automatic selection of compiler options using genetic techniques for embedded software design", *IEEE 14th Inter. Symposium on Comp. Intelligence and Informatics (CINTI)*, Budapest, Hungary, Nov. 19, (2013).

[24] Ornstein–Uhlenbeck process, https://en.wikipedia.org/wiki/Ornstein%E2%80%93Uhlenbeck_process, retrieved 1st Oct 2020.

[25] G. E. Uhlenbeck and L. S. Ornstein, "On the Theory of the Brownian Motion", Phys. Rev. **36**, 823, September 1930.

[26] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining Improvements in Deep Reinforcement Learning", arXiv:1710.02298v1 [cs.AI] 6 Oct 2017.

[27] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, I. Mordatch, "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments", arXiv:1706.02275v4 [cs.LG] 14 Mar 2020.

[28] W. Farag, "Multi-Agent Reinforcement Learning using the Deep Distributed Distributional Deterministic Policy Gradients Algorithm", *Intern. Conf. on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT'20)*, Bahrain, 20-21 Dec., (2020).