Self-Driving Car Agent in a Custom Environment using Deep Reinforcement Learning

Himadri Sankar Chatterjee
MCA
Vellore Institute of Technology
Vellore
himadri.sankar2019@vitstudent.ac.in

Megha Roy
MCA
Vellore Institute of Technology
Vellore
megha.roy2019@vitstudent.ac.in

Guided by, Varalakshmi M
Assistant Professor , *SITE*
Vellore Institute of Technology
Vellore
mvaralakshmi@vit.ac.in

Abstract—**Programming autonomy in robots and software agents has been an important topic of research for quite a few years, which has helped in development of various software agents such as autonomous driving agents. Reinforcement learning is a type of machine learning where an agent tries to fulfill a certain task in order to maximize the cumulative reward it receives while optimizing its actions. In our paper, we propose to use Deep Reinforcement Learning, an area that combines both Deep Learning and Reinforcement Learning, to help a car agent learn how to drive by itself. This project aims to train a car agent to navigate an uneven terrain, starting from an initial position, to reach the destination, which is, a passenger. We are going to build our proposed custom environment in Unity and apply Deep Reinforcement Learning algorithms such as Proximal Policy Optimization (PPO) and Advantage Actor Critic(A2C) algorithm to train our car agent.**

Keywords—reinforcement learning, unity, deep reinforcement learning, tensorflow, ml-agents, self-driving car, PPO, A2C.

I. Introduction

Autonomous Driving is the future trend in technology and developing these autonomous driving systems has been an active area of research that is being extensively studied by various research teams around the world. Developing a safe and secure self driving agent for a given environment forms the core component of this field. Recent advancement in machine learning techniques, especially in deep learning methods, have helped achieve new benchmarks in this field, while exceeding the limits set by humans. Organisations like Tesla have already implemented their autonomous driving systems in their cars, that are successfully piloting themselves through highways and busy traffic. Other than various supervised, unsupervised algorithms, that are powering the development of these self-driving cars, a recent trend has been in the application of several reinforcement learning techniques to build agents that perform far better than any existing technique.

Reinforcement Learning has played a major role in developing intelligent systems that outperforms human beings in various tasks. It has come a long way from AlphaGo that defeated Lee Sedol, the best Go player of all time, to the present day OpenAI Five, that has already been triumphant against the winners of the DOTA Championship. A recent paper from the Open AI research team about four independent agents playing the game of hide & seek has established a breakthrough in this field. The agents not only master their respective tasks but also start exploiting the game engine to find loopholes within the game mechanics. These results have fueled extensive research in the field of applying Reinforcement Learning techniques, starting from finance, to video games to robotics.

The concept behind reinforcement learning is quite simple to understand. The whole task of learning is modeled after the way humans learn how to calibrate good and bad behaviour in life. As in life, a child may touch a flame, realize that it burns him or her. This would discourage the child from putting his hand into the flame again. On the other hand, doing house chores may earn him a few treats from his parents. This positively reinforces the act of doing

house chores as the child knows he would get rewards for it. Fig 1.1, gives a general idea about reinforcement learning in general. The *agent* is an independent entity that represents a simple human being that will learn to perform a task. The *environment* is a model of the world, where the agent takes a particular action. Based on the action taken, the *state* (, or the current condition) of the environment changes into a *new state* and it gives out some *rewards* (, or feedback) to the agent that suggests whether the action it took resulted in a condition that is favourable to the agent or not.

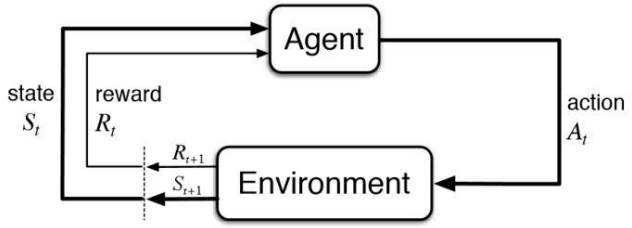


Fig. 1.: Reinforcement Learning

This paper attempts at developing an intelligent software car agent that can drive itself over an uneven terrain. The terrain is based in the form a hilly region, with slopes that are difficult to drive through and pits that should be avoided. Our agent has to navigate this terrain to arrive at its destination, which is a human being, located at some far-away point within the map. Developing this environment for our agent to learn from is an essential part of this project, for which we turned to Unity.

Unity is a game development engine that has been used by various game developers to build some of the greatest games of all time. Unity has built-in features that favoured the development of our terrain, along with the car and our destination, the human figure. Next, with the use of ml-agents, an open source Unity plugin, we define our academy, the agent actions and the brain of the agent. These enable games and simulations to serve or act as environments to train intelligent agents in. We then used a pre-implemented PPO algorithm from Unity ML-Agents and A2C using CNN to train the agent.

II. Literature Review

Fayjie et.al. in a 2018 paper titled 'Driverless Car: Autonomous Driving Using Deep Reinforcement Learning In Urban Environment'[2] presented a custom urban environment built in Unity Game Engine that navigated autonomously and avoided obstacles. The authors used Deep Q Network, a combination of Q learning and neural networks, to train the car. The car in the simulated environment had two sensors at its front- a laser sensor and a camera sensor. The authors tested a prototype of the proposed autonomous car using a camera sensor and Hokuyo Lidar Sensor embedded with a Nvidia Jetson TX2 embedded GPU. Their network architecture consisted of 3 convolutional layers and 4 dense layers. The camera sensor fed the neural network 4 frames at a time and the lidar sensor fed map plots to the neural network. The model then gave a probability distribution output of 5 action values- Left, Right, Keep Going, Accelerate and Brake. The final car prototype that was built was equipped to run DQN in real-time.

Güçkıran et.al. in a 2019 paper titled 'Autonomous Car Racing in Simulation Environment using Deep Reinforcement Learning'[3] tested two deep reinforcement learning algorithms to train the autonomous car agent- SAC and Rainbow DQN. Since developing a physical car agent model to test these algorithms is inefficient and risky, the authors opted for TORCS. TORCS or The Open Racing Car Simulator is a portable interface for artificial intelligence development. It is also flexible and open. The authors utilized the continuous action space to train the SAC model. This continuous action space was discretized to use the data to train the DQN model. The car agents were concluded to have completed the tracks with an average speed of around 140 km/h most of the time. The authors also concluded that the agents were able to function in a fairly generalized manner on unseen tracks. Overall, SAC algorithm seemed to have worked better than DQN since SAC promotes the maximization of entropy which leads to the agent exploring more uncertain action spaces. Also, SAC seemed to have performed better due to its ability to work with continuous action space.

Cai et.al. in a 2020 paper titled 'High-Speed Autonomous Drifting With Deep Reinforcement Learning'[4], the authors used CARLA, a simulator that provides different vehicles in a dynamic world. The algorithm of SAC, also known as Soft Actor Critic, had been used to teach the autonomous car how to drift. The car is trained on six different maps, each with varying levels of difficulty. The first map is relatively easy and allows the car agent to learn basic driving expertise. The car agent was also able to generalize well given varied levels of difficulty.

Patachi et.al. in a 2019 paper titled 'A Comparative Performance Study of Reinforcement Learning Algorithms for a Continuous Space Problem'[6] used the 'Mountain Car' environment to test two Reinforcement Learning algorithms which are Q learning and Deep Q learning. The authors trained the car using Q learning, Q learning with neural networks and Deep Q Network.

Qin et.al. in their 2019 paper titled 'Sim-to-real: Six-legged Robot Control with Deep Reinforcement Learning and Curriculum Learning' [5] created a six legged robot that was trained using Actor Critic network with PPO or proximal policy optimization.

IV. SIMULATED ENVIRONMENT AND AGENT DESIGN

The simulation environment has been designed in the Unity Game Engine. We used Unity standard assets to build and ml-agents toolkit to train our car agent in an uneven terrain. ML-Agents toolkit allows researchers and game developers to build and test machine learning algorithms, including reinforcement learning algorithms. In our project, for the purpose of training the car agent, we have created two environments with similar terrains but varying levels of difficulty. In our first environment, the terrain is comparatively smaller with 200x200 dimension along its x and z axis and it is slightly uneven. We use this environment to train our car agent in the initial stage to help it learn basic driving and maneuvering skills. After the car learns how to optimally reach the passenger, which is the goal state, in the first terrain, we transfer the trained model over to the second environment. The second environment is comparatively larger and has a more uneven terrain. The trained model is checked to see if it can generalize its learnt behaviour in a similar but a slightly more complex environment.

We attach a CameraSensor Component to our car agent. This allows the reinforcement learning model to accept the visual observations of the agent to learn optimized behaviour for efficient functioning of the agent as a self-driving car. The images passed as states to the brain is a 64X64 dimensional grayscale image, from the corresponding camera component. The visual observations are grayscale images as colours in the image do not provide any additional information that may optimize the reinforcement learning process. Rather, it adds unnecessary overhead. The reinforcement learning algorithm accepts the observed frames and outputs a gaussian distribution for action selection. From this distribution of continuous values, the model samples an action that dictates the agent's subsequent moves.

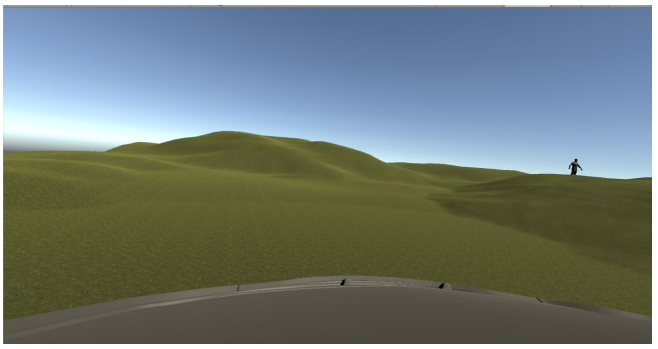


Fig 4.1:A single frame of the input image.

The reward function for the agent includes three main components:

1. A reward of **-0.04f** for each step taken.
2. It receives a reward of **-.6f** if the car falls off the platform.
3. On reaching the target, the final reward of **1.0f** is received.

The -0.04f reward at each step is to force the agent to take a step at each instance instead of staying ideal. However, the reward function is not universal for all training environments. Often for the large environment, the agent got stuck in a local minima, where the agent would drive the car backwards and fall off the platform, thus ending the episode with a cumulative reward, often less than the acquired while traversing the terrain randomly. We had to tweak the reward values a little for each different environment.

The car agent is imported from the Standard Assets of Unity. The movement of the car is defined on the values passed to it corresponding to the *steering* value and the *acceleration* value for the car. The *steering* value is passed by the horizontal arrow keys and the measure of the value is defined as:

$$h = \begin{cases} +1, & right\ tunr\ (right\ arrow\ key) \\ -1, & left\ turn\ (left\ arrow\ key) \end{cases}$$

The *acceleration* value is passed by the vertical arrow keys and its measure is given as:

$$v = \begin{cases} +1, & move\ forward\ (up\ arrow\ key) \\ -1, & move\ backward\ (down\ arrow\ key) \end{cases}$$

Each episode during the training of the agent terminates when one of the following condition is met:

1. The car agent falls off the terrains. Thus, if the y-component of the position metric of the car in the platform becomes less than 0, then we end the episode with a large negative reward.
2. The car reaches its target, If the agent arrives at any point with a radius of 30 units with the target at the center, the episode ends with a large positive reward.

V. DEEP REINFORCEMENT LEARNING ALGORITHMS

Proximal Policy Optimization[7]**:** This is currently the state-of-the-art reinforcement learning algorithm in use, introduced by researchers at OpenAI in the year 2017 in their paper titled 'Proximal Policy Optimization Algorithms'. The principle behind Proximal Policy Optimization, or PPO, is to steer clear from a substantial update of the policy. Proximal policy optimization is an on-line policy which essentially means that it does not have an experience replay buffer. The two important concepts introduced in this paper were- Clipped Surrogate Objective and the use of K number of epochs of gradient ascent for policy update.

PPO, similar to TRPO (Trust Region Policy Optimisation), opts to conservatively update the policy function by using the clipping surrogate objective:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right] \quad [7]$$

where $r_t(\theta)$ is the ratio of the current policy and the old policy function, $\hat{A}_t$ is the advantage function, that suggests how much better the chosen action at a given state was relative to the baseline estimate of taking that action at that state and ϵ is a hyperparameter generally taken as 0.2, is used to balance the rate of exploration and exploitation while training the agent. To ensure that the update does not make the policy function stray too far from the current policy function, $L^{CLIP}(\theta)$ takes the minimum of $r_t(\theta)$ and a clipped value of $r_t(\theta)$ in the range of ~(0.8, 1.2). This ensures that if an action is extremely good or extremely bad, it does not update the policy function abruptly. Here is the pseudocode:

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Fig. 5.1: The PPO algorithm, Actor-Critic Style [7]

Advantage Actor Critic: A2C, also known as Advantage Actor Critic algorithm utilizes two networks: a value based Critic network and a policy based Actor Network. The Actor network controls the actions taken by the agent while the Critic network quantifies how good the action is. The Actor and Critic neural networks have their own set of parameters that are updated separately.

Algorithm 1 Advantage actor-critic - pseudocode

// Assume parameter vectors θ and θ_v
Initialize step counter $t \leftarrow 1$
Initialize episode counter $E \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta)$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t+1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta_v) & \text{for non-terminal } s_t \text{ //Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ: $d\theta \leftarrow d\theta + \nabla_\theta \log \pi(a_i|s_i;\theta)(R - V(s_i;\theta_v)) + \beta_e \partial H(\pi(a_i|s_i;\theta))/\partial\theta$
    Accumulate gradients wrt θ_v: $d\theta_v \leftarrow d\theta_v + \beta_v(R - V(s_i;\theta_v))(\partial V(s_i;\theta_v)/\partial\theta_v)$
  end for
  Perform update of θ using $d\theta$ and of θ_v using $d\theta_v$.
  $E \leftarrow E+1$
until $E > E_{max}$

Fig 5.2: A2C Algorithm pseudocode.

The A2C model as depicted in the given image helps in understanding the algorithm better,

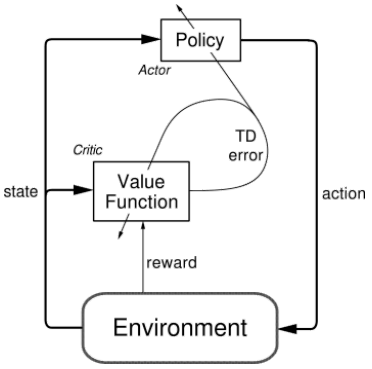


Fig 5.3: The A2C Algorithm

The Actor Network or the Policy Network receives a single frame of visual observation from the environment as the state and determines the best action to take from there. The state returned by the environment is a 64X64 dimensional grayscale image of the environment as seen from the car. We defined a convolutional neural network for the Actor Network with two output layers, viz. the mu value and the sigma value for the normal distribution used to estimate the actions for that state. The loss function for the policy is defined as:

$$\Delta\theta = \alpha \, \nabla_\theta(log \, \pi_\theta(s,a))\hat{q}_w(s,a)$$

The Critic Network or the Value Network plays the role of a supervisor, where it determines how good the action taken for the given state is, by returning a score for the same. We have defined a convolutional neural network similar to the Actor

Network model to estimate the value of the action taken. The value loss function is defined as:

$$\Delta w = \beta\left(R(s,a) + \gamma \hat{q}_w(s_{t+1}, a_{t+1}) - \hat{q}_w(s_t, a_t)\right)\nabla_w \hat{q}_w(s_t, a_t)$$

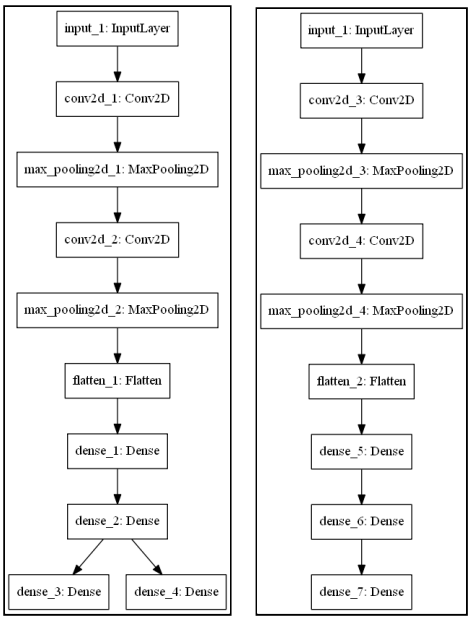


Fig 5.4: Actor CNN Model Critic CNN Model

A2C, on which algorithms like PPO have been built, also uses the advantage function A to measure how good the chosen action at a time step is compared to the estimated baseline reward of taking an action in a given state. In A2C, we can have multiple versions of the same environment. The global parameters of the network are updated synchronously by averaging the gradients of each network. Due to the constraint on the computation power available to us, we decided to work on a single environment.

VI. TRAINING AND RESULTS

We started to test our car agent in the two environments: a smaller, slightly more even terrain and a larger uneven terrain. We first attempt to train our agent using the inbuilt PPO algorithm provided in Unity ML-Agents. PPO processes these visual observations and provides a continuous action space output that tells the car agent how to move horizontally and vertically. Since we are using continuous action values and not discrete values, it provides a smoother movement of our car agent. The agent is able to drive towards the target after a few hundreds of episodes. The training took around 5 hours on our computers.

The car agent learns basic driving skills on the easier terrain after which the trained model is associated with the car agent in the larger uneven environment.

The policy loss and value loss graphs after training the car using PPO algorithm are:

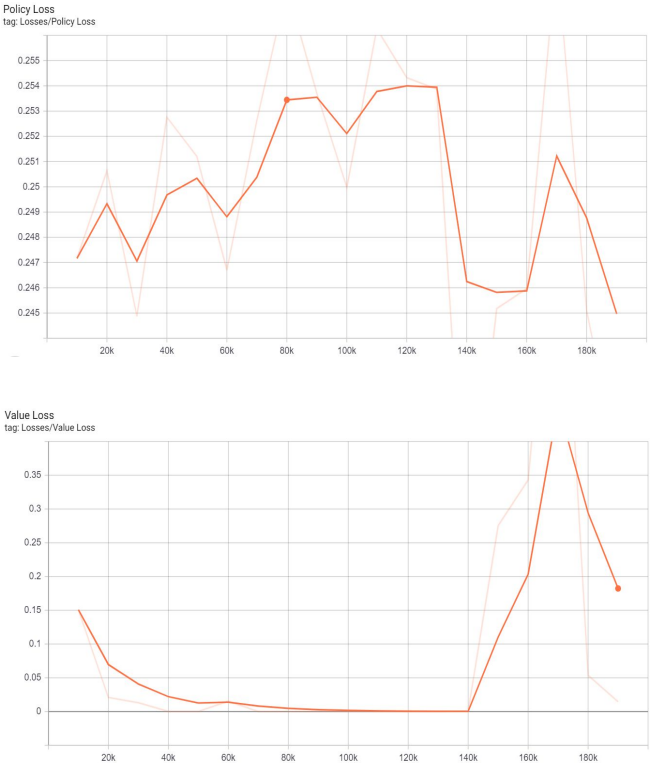
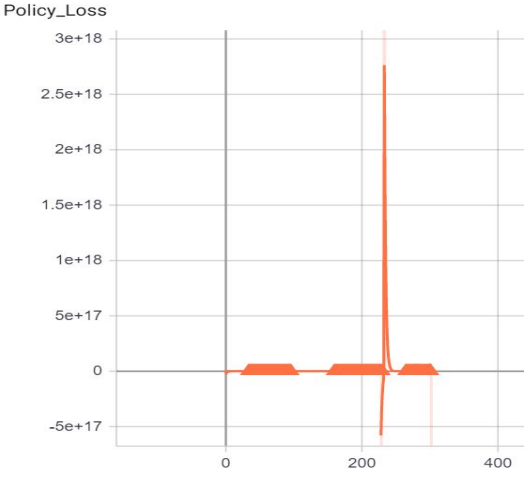


Fig 6.1: The Policy Loss and Value Loss plot for PPO

We then applied the A2C algorithm defined in Python. We used the **gym-unity wrapper** library to interact with the environment from outside the Unity interface. Although the complete training of the model was not successful, the model eventually converged to some local minimas to complete the episodes. We could not continue the training due to memory constraints. After around 300 episodes of training the Policy Loss and the Value Loss of the model is recorded as:



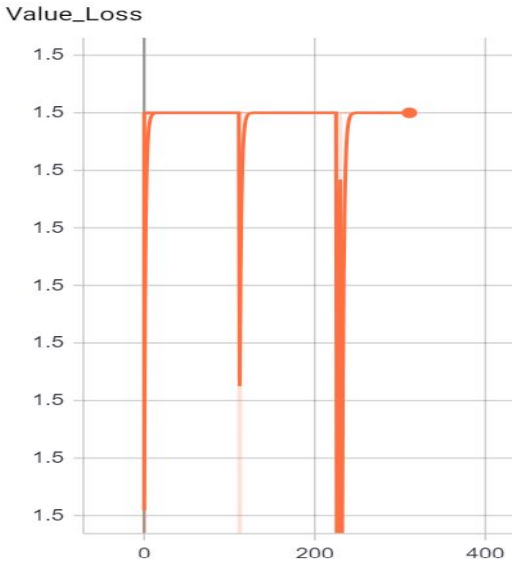


Fig 6.2: Policy Loss and Value Loss plot for A2C

The camera associated with the car agent sends one frame per time step to the model. The frame is processed individually by the CNN architecture defined in the Actor network and the Critic network. In the Actor network, the output of the second dense layer is sent to two different dense layers, each of which outputs two values corresponding to the action space. These values are used to find a gaussian distribution from which an action is chosen for the horizontal and vertical movement of the agent. On the other side, in the Critic network, the output of the second dense layer is sent to a third dense layer which gives us two output values that act as an estimated baseline value of taking an action in a given state. This value is used to calculate the advantage function and update the value function of the Critic network and the policy of the Actor network.

VII. CONCLUSION

We have trained our car agent in a custom environment built using Unity Game Engine with the help of two deep reinforcement learning algorithms. Using the in-built PPO algorithm in Unity ML-Agents, we trained the car in a small and a larger environment. The car agent learned to drive in the smaller environment and reach its target within 5 hours of training. In the larger environment, it took the car agent 21 hours to learn how to reach the target efficiently. Using the A2C architecture, the car agent was trained in the larger environment and took 9 hours to learn efficient driving and maneuvering skills in the smaller environment.

Due to the constraint on the computation power available to us, we tried to train the agent using only CPU's. The results are quite inefficient. However the approach at generalising the training of the car agent, starting from a small environment and then shifting the focus to larger and complex environments with the knowledge gained, helped in faster initial training. Moving forward, we intend to insert more complexity in the environment with obstacles and differing land-types and employ machines with larger computation capacity to train the agent.

VIII. REFERENCES

1. Juliani, Arthur, et al. "Unity: A general platform for intelligent agents." *arXiv preprint arXiv:1809.02627* (2018).
2. A. R. Fayjie, S. Hossain, D. Oualid and D. Lee, "Driverless Car: Autonomous Driving Using Deep Reinforcement Learning in Urban Environment," 2018 15th International Conference on Ubiquitous Robots (UR), Honolulu, HI, 2018, pp. 896-901.
3. K. Güçkıran and B. Bolat, "Autonomous Car Racing in Simulation Environment Using Deep Reinforcement Learning," 2019 Innovations in Intelligent Systems and Applications Conference (ASYU), Izmir, Turkey, 2019, pp. 1-6.
4. Cai, Peide & Mei, Xiaodong & Tai, Lei & Sun, Yuxiang & Liu, Ming. (2020). High-Speed Autonomous Drifting With Deep Reinforcement Learning. IEEE Robotics and Automation Letters. PP. 1-1. 10.1109/LRA.2020.2967299.
5. B. Qin, Y. Gao and Y. Bai, "Sim-to-real: Six-legged Robot Control with Deep Reinforcement Learning and Curriculum Learning," 2019 4th International Conference on Robotics and Automation Engineering (ICRAE), Singapore, Singapore, 2019, pp. 1-5.
6. A. Patachi and F. Leon, "A Comparative Performance Study of Reinforcement Learning Algorithms for a Continuous Space Problem," 2019 23rd International Conference on System Theory, Control and Computing (ICSTCC), Sinaia, Romania, 2019, pp. 860-865.
7. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *ArXiv, abs/1707.06347*.
8. Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning."

International conference on machine learning. 2016.
9. Wu, Yuxin, and Yuandong Tian. "Training agent for first-person shooter game with actor-critic curriculum learning." (2016).
10. Kaelbling, Leslie Pack, Michael L. Littman, and Andrew W. Moore. "Reinforcement learning: A survey." *Journal of artificial intelligence research* 4 (1996): 237-285.
11. Sutton, Richard S., and Andrew G. Barto. *Introduction to reinforcement learning*. Vol. 135. Cambridge: MIT press, 1998.
12. Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).
13. Nandy, Abhishek, and Manisha Biswas. "Unity ML-Agents." *Neural Networks in Unity*. Apress, Berkeley, CA, 2018. 27-67.
14. Lillicrap, Timothy P., et al. "Continuous control with deep reinforcement learning." *arXiv preprint arXiv:1509.02971* (2015).