An Analysis of Continuous Action Reinforcement Learning Algorithms

CS7IS2 Project (2019-2020)

Andrew Hynes, Eimear Cosgrave, Colin Comey, Sophie Hegarty

Hynesa1@tcd.ie, ecosgrav@tcd.ie, comeyc@tcd.ie, hegartso@tcd.ie

Abstract. In recent years there has been a substantial amount of research focused on developing reinforcement learning algorithms for continuous action spaces. This document compares three of these approaches, namely: Deep Deterministic Policy Gradients, Proximal Policy Optimization and Augmented Random Search. These three algorithms represent different approaches to updating neural networks to obtain an optimal policy in a reinforcement learning setting. The algorithms were compared on two different OpenAI Gym environments and were in terms of overall performance. Results showed that the algorithms performance was highly dependent on the environment in which it was deployed.

1 Introduction

Reinforcement Learning (RL) has begun to garner a lot of attention recently, due to its ability to learn effectively and independently in a wide variety of tasks. RL itself represents a wide variety of algorithms which are capable of training agents to behave optimally in an environment and maximise a reward.

However, whilst reinforcement learning has been around for decades, with many popular algorithms such as Q-Learning dating back as far as 1989 [1], it is only recently that powerful and robust algorithms have been designed to deal with continuous action spaces.

The evolution of deep learning methods has played a key role in the development of these continuous action algorithms, with neural networks now being used to define a RL's agent parameters. This has been made possible with new algorithms for updating these neural networks, with these algorithms generally falling into one of three families: policy gradient methods, trust region methods and random search methods. In this research, in order to try gain insight to the most effective family of algorithms, the state of the art of each family will be tested on several OpenAI gym environments and compared against the other algorithm types.

This document is structured as follows. Section 2 will outline related work and discuss the variety of algorithms designed to train neural networks to operate in continuous action spaces. Section 3 will describe the problem and the 3 algorithms being analysed. Section 4 will discuss the experimental results, whilst section 5 will provide the conclusion to our work.

2 Related Work

In order to train neural networks to behave optimally in an environment, algorithms need to be created to allow for the efficient and stable update of the parameters of these networks. To this end a wide variety of approaches have been proposed to do this. These algorithms fall into three categories: policy gradient algorithms, trust region algorithms and random search algorithms.

The first policy gradient algorithm was proposed in [2] and is known as an actor critic model. It works by mapping states to a probability distribution of actions.

[3] proposed an alternative to the above by replacing the Q-value with an Advantage-value which incorporated a baseline. This algorithm became known as the Advantage Actor Critic or A2C. It was also shown that it can be extended to an asynchronous version known as Asynchronous Advantage Actor Critic or A3C.

[4] proposes Deep Deterministic Policy Gradient (DDPG) which is a deterministic version of the actor critic model proposed in [2].

Trust region methods were proposed in [5] with the development of an algorithm known as Trust Region Policy Optimization (TRPO). This algorithm is still very similar to the traditional policy gradient methods however updates are restricted to a trusted region using a hard constraint which is calculated using Kullback-Leibler divergence which describes the variation between two probabilistic distributions.

[6] combines TRPO with other features including experience replay and is known as the actor critic with experience replay algorithm (ACER).

[7] proposes an algorithm known as Actor Critic using Kronecker-Factored Trust Region (ACKTR), which used Kronecker-factored approximate curvature with trust regions and was highly scalable.

[8] proposed PPO which used a clipping function to stay within a trusted region. This removes the hard constraint in favour of a punishment.

The final category of algorithms is known as random search and several of these were first proposed in [9]. These algorithms explore in the parameter space as opposed to the action space. Essentially these algorithms work by randomly updating the policies

parameters and progressing in the direction which shows the most promise. The paper introduces a number of additional features to create Augmented Random Search.

The goal of this research is to compare the performance of the different family of algorithms and to this end it will focus on the most powerful and popular algorithms from each family of methods, namely: DDPG, PPO and Augmented Random Search. The DDPG algorithm is one of the most commonly used and most powerful policy gradient methods available and has been deployed in a wide variety of tasks ranging from robotic control [10] to autonomous driving [11]. PPO is also a very powerful algorithm with several benefits when compared to other algorithms such as TRPO, however, its popularity is mostly the result of its simple implementation when compared to other trust region methods. Nonetheless, it has still be successfully applied to a variety of tasks, including unmanned aerial vehicle (UAV) autopilot control in [12]. Finally, the augmented random search algorithm will also be tested. This algorithm claims to be superior to other reinforcement learning algorithms in terms of performance with the authors also arguing that it is also at least 15 times more efficient than the fastest competing alternative methods. All three of these algorithms can be considered the state of the art in reinforcement learning.

3 Problem Definition and Algorithm

The objective of this work is to analyse and compare the performance of different reinforcement learning algorithms in continuous state continuous action spaces. As outlined in section 1, continuous action environments rely on deep neural networks to describe their policy and as such require algorithms capable of efficiently updating the neural network's parameters in a stable fashion.

For this research three main families of network update algorithms have been identified, namely: policy gradient methods, trust region methods and random search methods. This research focuses on the state of the art algorithms in each of these families: DDPG, PPO and Augmented Random Search.

3.1 Deep Deterministic Policy Gradient (DDPG)

DDPG was first proposed in [4] and represented the first implementation of the deterministic policy gradient, something which had previously been considered to be impossible.

The algorithm makes use of the actor critic architecture, which means it is composed of two sections, an actor and a critic section. It is an off policy algorithm. The critic is responsible for learning a Q-value prediction, which describes the expected long term reward of taking a particular action in a particular state. The critic can be trained using the backpropagation algorithm on the loss function as seen in Fig. 3.1. Whilst the actor is updated based on the sampled policy gradient.

The algorithm also makes use of target networks and experience replay to improve sample efficiency and stability during learning. The pseudocode can be seen in Fig. 3.1.

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

Fig. 3.1

3.2. Proximal Policy Optimization (PPO)

[8] proposes Proximal Policy Optimization (PPO) which is a trust region policy optimization algorithm. It constrains the updates of the policy to within a certain region, which means the state and action trajectories are relatively consistent, allowing for more stable training. However, it still operates in a very similar way to basic policy gradient methods.

To do this PPO proposes adding a punishment onto the normal loss function which discourages updates outside a trusted region. This differentiates it from other trust region methods which impose a hard boundary. As a result, PPO can use a first order optimizer such as gradient descent, making it far more scalable than alternatives. This is done by using a surrogate loss function with a clip function, $\mathcal{L}^{Clip}_{\theta_k}(\theta)$, as seen in Fig. 3.2. By applying this surrogate loss function, large policy changes are discouraged, and stability can be improved. The algorithm can be seen in Fig. 3.2

Algorithm 5 PPO with Clipped Objective

Input: initial policy parameters θ_0, clipping threshold ϵ
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
  Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
  Compute policy update

$$\theta_{k+1} = \arg\max_\theta \mathcal{L}^{CLIP}_{\theta_k}(\theta)$$

  by taking K steps of minibatch SGD (via Adam), where

$$\mathcal{L}^{CLIP}_{\theta_k}(\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_k}\left[\sum_{t=0}^{T}\left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t^{\pi_k})\right]\right]$$

end for

Fig. 3.2

3.3 Augmented Random Search

Augmented Random Search was first proposed in [9] and is built on the basic random search algorithm which randomly updates the parameters of the policy in order to learn. As such, the algorithm explores the parameter space as opposed to the action space, which differentiates if from other methods.

To do this the algorithm relies on a hyperparameter, N, which describes the number of directions to sample at each iteration. Random noise is then added to the parameters in the positive and negative of each of these directions to make new test policies. These new policies are then rolled out over a set of states until a horizon of fixed length is reached. The rewards are gathered and then the policy is finally updated in the direction of best improvement, as can be seen in Fig 3.3.

This update function scales the update steps by the standard deviation of rewards, preventing updates getting progressively larger. The state space vector is also normalized. The pseudo code can be seen below in Fig. 3.3.

Algorithm 2 Augmented Random Search (ARS): four versions **V1**, **V1-t**, **V2** and **V2-t**

Hyperparameters: step-size α, number of directions sampled per iteration N, standard deviation of the exploration noise ν, number of top-performing directions to use b ($b < N$ is allowed only for **V1-t** and **V2-t**)

Initialize: $M_0 = \mathbf{0} \in \mathbb{R}^{p \times n}$, $\mu_0 = \mathbf{0} \in \mathbb{R}^n$, and $\Sigma_0 = \mathbf{I}_n \in \mathbb{R}^{n \times n}$, $j = 0$.

while ending condition not satisfied **do**

Sample $\delta_1, \delta_2, \ldots, \delta_N$ in $\mathbb{R}^{p \times n}$ with i.i.d. standard normal entries.

Collect $2N$ rollouts of horizon H and their corresponding rewards using the $2N$ policies

$$\textbf{V1:} \begin{cases} \pi_{j,k,+}(x) = (M_j + \nu\delta_k)x \\ \pi_{j,k,-}(x) = (M_j - \nu\delta_k)x \end{cases}$$

$$\textbf{V2:} \begin{cases} \pi_{j,k,+}(x) = (M_j + \nu\delta_k)\,\mathrm{diag}\,(\Sigma_j)^{-1/2}(x - \mu_j) \\ \pi_{j,k,-}(x) = (M_j - \nu\delta_k)\,\mathrm{diag}(\Sigma_j)^{-1/2}(x - \mu_j) \end{cases}$$

for $k \in \{1, 2, \ldots, N\}$.

Sort the directions δ_k by $\max\{r(\pi_{j,k,+}), r(\pi_{j,k,-})\}$, denote by $\delta_{(k)}$ the k-th largest direction, and by $\pi_{j,(k),+}$ and $\pi_{j,(k),-}$ the corresponding policies.

Make the update step:

$$M_{j+1} = M_j + \frac{\alpha}{b\sigma_R} \sum_{k=1}^{b} \left[r(\pi_{j,(k),+}) - r(\pi_{j,(k),-}) \right] \delta_{(k)},$$

where σ_R is the standard deviation of the $2b$ rewards used in the update step.

V2 : Set μ_{j+1}, Σ_{j+1} to be the mean and covariance of the $2NH(j+1)$ states encountered from the start of training.[2]

$j \leftarrow j + 1$

end while

.

Fig 3.3

4 Experimental Results

4.1 Methodology

OpenAI provide standardised environments to allow easy comparison between reinforcement learning algorithms. In this report the three algorithms mentioned in Section 3 were trained on 2 continuous OpenAI environments. Those being MountainCarContinuous-v0 and Pendulum-v0. MountainCar involves a car between

two hills with the goal of driving up the mountain using the optimum momentum. The pendulum environment involves swinging a pole around a joint with the goal to make it stay upright. These environments were chosen due to the varying degree of difficulty associated with them. The pendulum environment can be considered relatively trivial with the reward being the distance from the vertical upright position in degrees. This gives a maximum reward of zero and a negative reward based on the distance between the current position and the upright position. The mountain car is more challenging with a reward of 100 for reaching the target with the squared sum of actions being subtracted. This means if the square sum of actions is greater than 100, total reward will be less than zero. As such a local minimum exists where the agent will choose not to take any action throughout the episode. The mountain car is also very difficult as there is no indication from the reward if the agent is getting closer to the goal. Therefore, this environment can be considered a type of sparse reward environment.

Each algorithm was optimized through iterative parameter tuning and then used to train an agent in each of the two environments. The mean reward per episode was recorded for comparison. They were run 10 times on each environment and an average of these was taken to compare with the other algorithms.

4.2 Results

DDPG Results:

Figure 4.1 shows how the DDPG algorithm learnt in the 'Pendulum-v0' environment. Despite never reaching the maximum reward of 0, the algorithm learns to adapt its weights reasonably well to the environment. After some brief exploration at the start which lead to a dip in rewards, it learns to explore in the direction that gives better results. It does so very effectively, which is indicated by the steep slope between episode 20 and episode 40.

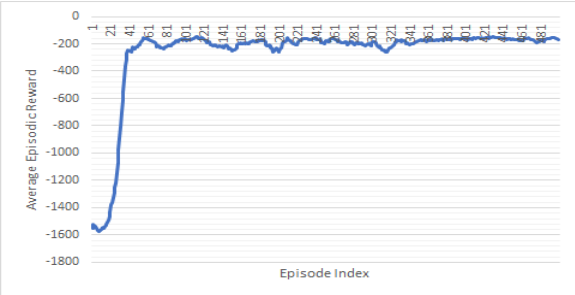

Figure 4.1: Average episodic rewards using the DDPG algorithm in the 'Pendulum-v0' environment.

Figure 4.2 shows how the DDPG algorithm learns in the 'MountainCarContinuous-v0' environment. Due to the added difficulty of the environment it requires a lot of exploration to achieve the reward.

Throughout the first 40 episodes, the algorithm tries exploring the action space in order to achieve a better reward. However, the car never reaches the goal state within

a single period. Thus, it only experiences the penalty term despite it potentially getting closer to the goal state. The algorithm then learns to maximise this by simply limiting the size of the actions taken. Note that the rewards are always negative, indicating that the car never reaches the goal state.

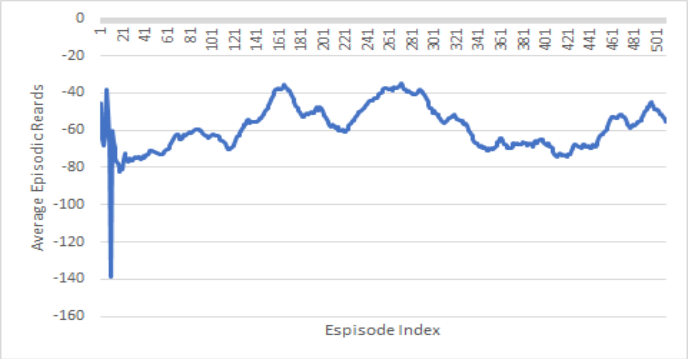

Figure 4.2: Average episodic rewards using the DDPG algorithm in the 'MountainCarContinuous-v0' environment.

PPO results

The PPO algorithm was capable of training quite effectively in the Pendulum environment. However, it was unable to achieve the maximum reward of 0, instead reaching a maximum of -139. The mean rewards, as seen in Figure 4.6, were noisy. The range of this was reduced by changing hyperparameters, including reducing the learning rate to 3e-4 and increasing the number of epochs from 2 to 10. This gave a cleaner curve, even though there is a local minimum that the agent gets stuck in for a number of episodes.

However, PPO struggled to train on the mountain car environment, with all rewards being negative. This was the same even after hyperparameter tuning. This is most likely due to the environment using sparse rewards which is particularly troublesome for an on-policy algorithm like PPO which discards experiences after each update. Even if the agent randomly got a positive reward, it is unlikely that a single policy gradient update would be enough to push the agent into a consistent positive reward. However, similar problems were experienced by DDPG, an off policy algorithm, which may mean that the problem relates to the use of policy gradients.

For PPO, it was found that this can be overcome with curiosity [13], this uses a mechanism that creates an intrinsic reward which rewards the model for exploring new territories. As we can see in figure 4.3, the PPO algorithm with curiosity trains much better, and reaches a higher maximum than without it.

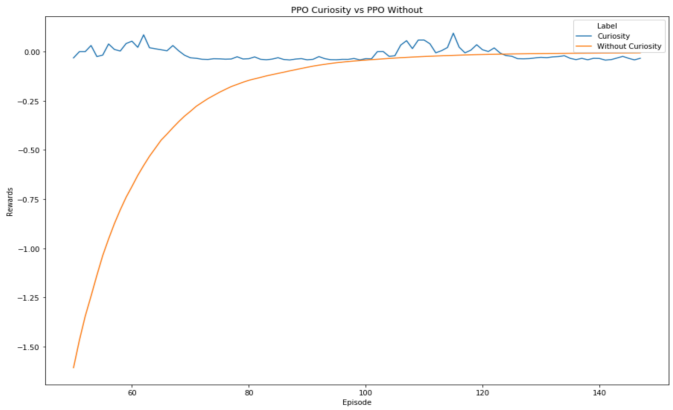

Figure 4.3 Mean reward per episode for PPO with and without curiosity on MountainCarContinuous

ARS results

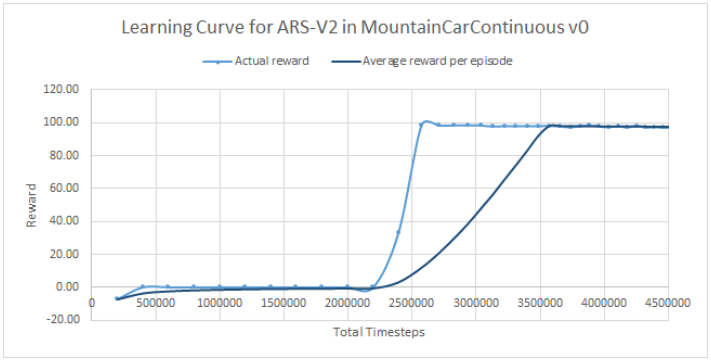

Figure 4.4: Results of ARS-V2 for MountainCarContinuous-v0.

In contrast to PPO and DDPG, ARS performed very well on the Mountain Car environment, where it learnt very quickly that staying still can give a reward of 0 which appears optimum. However, after enough timesteps it finds the target and extremely quickly learns the optimum policy to reach this target.

However, it was found that ARS was very sensitive to hyper-parameters and the agent was likely to remain in the local maximum of 0 if the learning rate or number of directions considered was too low. These hyperparameters were tuned and Figure 4.4 show the results of ARS-V2 using a learning rate of 0.01 and the number of directions equalling 100 with only the top 50 directions (by rollout reward) used for updating the policy.

The leader board for the Mountain Car Continuous environment is based on number of episodes need to reach a score of 90. Using this evaluation metric, the ARS-V2 algorithm solves the task after 13 episodes placing it third in the overall score board.

Interestingly, ARS was less successful in the Pendulum-v0 environment. The hyper-parameters were systematically altered without any impact on results. The agent did not end up learning a policy to perform well in this environment, no matter the hyper-parameter combination. The learning rate was varied from 0.00001 to 0.05, the number of directions investigated varied from 50 to 250, and the number of 'best' directions used to calculate the policy update was varied from 25 to 250, based on the author's suggestions and further research into similar problems. The number of total timesteps for training was expanded to see if the agent just learnt extremely slowly, but this showed no benefit and eventually experiments were taking over 4 hours, significantly longer than the other two algorithms.

The results were quite noisy but the average reward per episode shows a slight trends. Though there is the presence of an upward trend, insinuating the agent is optimising its policy, these rewards do not ever reach higher than –1100 which is not an impressive result in this environment. Peaks are followed by drops in an almost cyclical manner.

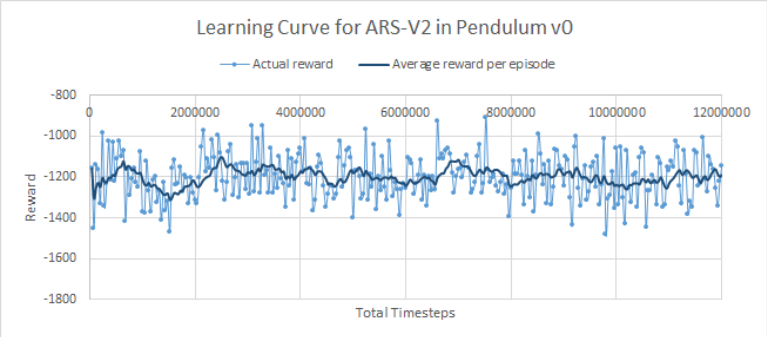

Figure 4.5: Results of ARS-V2 for Pendulum-v0.

The performance of each algorithm plotted against each other can be seen in Figure 4.6. Note: the legend varies between the two plots

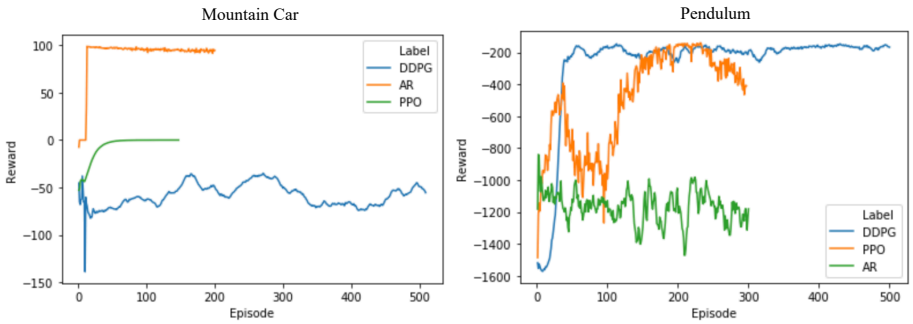

Figure 4.6, Mean reward per episode run on the MountainCarContinuous and Pendulum environments. Note: the Legend varies between the two plots

5 Conclusions

The results from this research can be best described as varied. Whilst the ARS algorithm was substantially better in the Mountain Car environment than the other two algorithms, it struggled greatly in the simpler Pendulum environment. In contrast, the two algorithms that struggled greatly in the Mountain Car environment, PPO and DDPG, performed well in the Pendulum environment. This could be due to the intrinsic nature of the updates each algorithm use. DDPG and PPO rely on gradients and hence direction to update their parameters, however, in sparse reward settings this direction can be difficult to find as a positive reward is only received when the goal is achieved. This may also explain why curiosity helped PPO, by introducing more reward. However, ARS overcomes this problem by removing the reliance on direction, instead preferring to randomly try multiple directions. As such, it is more successful in a sparse setting. However, this approach then seems to hinder its approach in the pendulum environment where the reward function can be easily approximated and used to provide direction through gradient updates. This is how PPO and DDPG work and may explain why they are so successful here. Therefore, it can be concluded that performance is not reliant solely on the algorithm or family of algorithms used, but also the environment in which the algorithm is deployed. Therefore, it cannot be definitely concluded which algorithm is best.

References

1. Watkins, Christopher John Cornish Hellaby. "Learning from delayed rewards." (1989).
2. Samuel, Arthur L. "Some studies in machine learning using the game of checkers." *IBM Journal of research and development* 3.3 (1959): 210-229
3. Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning." *International conference on machine learning*. 2016.
4. Lillicrap, Timothy P., et al. "Continuous control with deep reinforcement learning." *arXiv preprint arXiv:1509.02971* (2015).
5. Schulman, John, et al. "Trust region policy optimization." *International conference on machine learning*. 2015.
6. Wang, Ziyu, et al. "Sample efficient actor-critic with experience replay." *arXiv preprint arXiv:1611.01224* (2016)
7. Wu, Yuhuai, et al. "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation." *Advances in neural information processing systems*. 2017.
8. Schulman, John, et al. "Proximal policy optimization algorithms." *arXiv preprint arXiv:1707.06347* (2017).
9. Mania, Horia, Aurelia Guy, and Benjamin Recht. "Simple random search provides a competitive approach to reinforcement learning." *arXiv preprint arXiv:1803.07055* (2018).
10. Danel, Marek. "Reinforcement learning for humanoid robot control." *POSTER, May* (2017).
11. Guo, Fenggen, and Zizhao Wu. "A Deep Reinforcement Learning Approach for Autonomous Car Racing." *International Conference on E-Learning and Games*. Springer, Cham, 2018.
12. Koch, William, et al. "Reinforcement learning for UAV attitude control." *ACM Transactions on Cyber-Physical Systems* 3.2 (2019): 1-21.
13. Pathak, D., Agrawal, P., Efros, A.A. and Darrell, T., 2017. Curiosity-driven exploration by self-supervised prediction. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (pp. 16-17).