Implementation of Deep Reinforcement Learning Algorithms on Racetrack Environment

Aniket Patil
Robotics Engineering Department
Worcester Polytechnic Institute
Worcester, USA
apatil2@wpi.edu

Prathamesh Bhamare
Robotics Engineering Department
Worcester Polytechnic Institute
Worcester, USA
pbhamare@wpi.edu

Rutwik Bonde
Robotics Engineering Department
Worcester Polytechnic Institute
Worcester, USA
rrbonde@wpi.edu

***Abstract*—In this paper, we study and compare three different deep reinforcement learning policy gradient algorithms. We implement Proximal Policy Optimization (PPO), Deep Deterministic Policy Gradient (DDPG), and Asynchronous Advantage Actor Critic (A3C) algorithm on the *racetrack-v0* in the 'Highway Environment' from OpenAI Gym. Further, we analyze the behavior of the algorithms by comparing them to each other on the bases of rewards they achieve, the time they take to train, and the ease of their implementation. On the basis of the comparison that we make, it is observed that PPO algorithm performs better than DDPG and A3C algorithm during testing. Training time for A3C algorithm is lesser as compared to PPO and DDPG.**

***Index Terms*—Policy Gradient, Proximal Policy Gradient (PPO), Deep Deterministic Policy Gradient (DDPG), Asynchronous Advantage Actor Critic (A3C), Racetrack Environment.**

Link to our GitHub repository: https://github.com/aniketmpatil/RL-Highway-Env-Project

I. Introduction

Today, major automobile industries are trying to achieve true autonomous driving. The main motivations behind the idea are safer roads, increase in productivity, more economical transportation, and efficient movement of vehicles. Following this motivation we try to address two of the most challenging tasks to achieve true autonomy: maintaining the lane, and obstacle avoidance while handling interactions with the other vehicles.

We try to implement and compare three deep reinforcement learning policy gradient algorithms (PPO, DDPG, A3C) on the Racetrack from the highway environment. Racetrack is a continuous control environment, where the agent has to follow the tracks while avoiding collisions with other vehicles, as shown in figure 1. The yellow box in the figure is the agent whereas the blue box are other vehicles that serve as moving obstacles on the white lane.

II. Overview of Policy Gradient Methods

Reinforcement Learning is a framework in which reward related learning problems of machines, humans, or animals can be phrased. However, most of the methods proposed in the reinforcement learning community are not yet applicable to many problems such as robotics, self driving cars, motor controls, etc. This may result from problems with uncertain state information. Thus, those systems need to be modeled as partially observable Markov decision problems which often results in excessive computational demands, and time complexities. Most traditional methods have no convergence guarantees and there also exist a few divergence examples. Moreover, continuous states and actions in high dimensional spaces cannot be treated by most off the shelf reinforcement learning approaches.
Policy gradient methods differ significantly as they do not suffer from these problems in the same way. Sometimes, uncertainty in the state might degrade the performance of

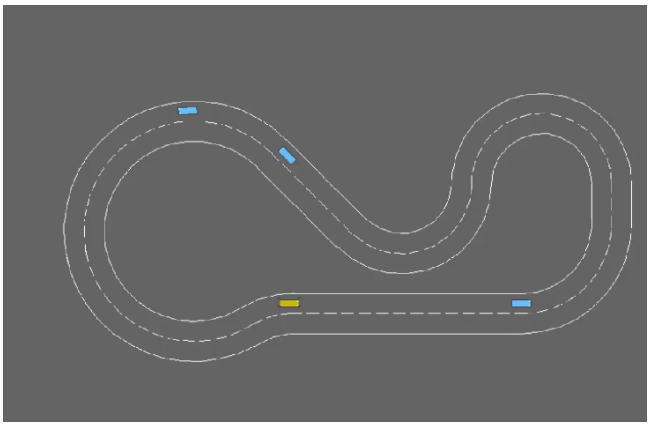

Fig. 1. Racetrack Environment

Action Space:	State observation:	Reward Space:
a. Steering angle 1. Continuous = [-pi/4, pi/4] 2. Discrete = [-0.5, 0, 0.5]	a. Occupancy grid= [-18, 18] b. Grid-step = [2,2] c. Features = ['presence', 'on-road']	a. Collision reward = -5 b. Action reward = 0.3 c. Offroad penalty = -1 d. Lane centering = 4 e. Subgoal reward ratio = 1

Fig. 2. Environment Configuration

the policy but the optimization techniques for the policy do not need to be changes. Continuous state and actions can be dealt with an exactly the same way as discrete ones while, in addition, the learning performance is often increased. Convergence at least to a local optimum is guaranteed. The advantages of policy gradient methods for real world applications are numerous. Among the most important ones are that the policy representations can be chosen so that it is meaningful for the task and can incorporate domain knowledge, that often fewer parameters are needed in the learning process than in value function based approaches and that there is a variety of different algorithms for policy gradient estimation which have a rather strong theoretical underpinning. Additionally, policy gradient methods can be used either model-free or model-based as they are a generic formulation.

Due to the advantages of policy gradient methods, they have become particularly interesting for robotics and self driving car applications as these have both continuous action and states. We therefore try to implement and compare three deep reinforcement learning policy gradient algorithms (PPO, DDPG, A3C) on the Racetrack from the highway environment. Based on the comparison of these algorithms we try to find the best policy gradient algorithm that can be applied to real life self driving scenarios.

III. Proximal Policy Optimization (PPO)

Proximal Policy Optimization is a policy gradient method for reinforcement learning, which alternate between sampling data through interaction with the environment, and optimizing a "surrogate" objective function using stochastic gradient ascent. The standard policy gradient methods perform one gradient update per data sample, whereas the objective of proximal policy optimization (PPO) enables multiple epochs of minibatch updates. The most common implementation of PPO is via the Actor-Critic Model which uses 2 Deep Neural Networks, one taking the action (actor) and the other handles the rewards (critic), as shown in figure 2.

A. Clipping objective function in PPO

The objective function that is optimized in PPO is an expectation, which means it is computed over a batches of trajectories. The expectation operator is taken over the minimum of two terms, the normal policy gradients objective and the clipped version of the normal policy gradients objective.

Clipped Objective:

$$L^{CLIP}(\theta) = E_t[min(r_t(\theta).A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon).A_t)]$$

where:

- Theta is the policy parameter
- E denotes the empirical expectation over timesteps
- r denotes the ratio of the probabilities under the new and old policies respectively (sampling ratio)

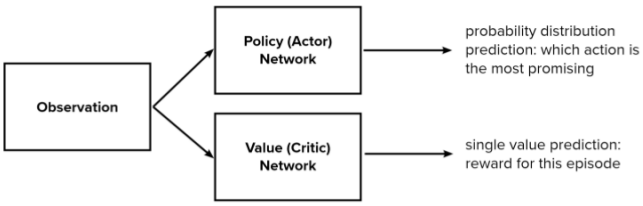

Fig. 3. PPO Network

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. 4. PPO Algorithm

- A is the estimated advantage at time t
- Epsilon is a hyperparameter, usually 0.2

Final PPO Objective:

$$L_t^{CLIP+VF+S}(\theta) = E_t[L^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + C_2 S[\pi_\theta](s_t)]$$

As we use a neural network architecture that shares parameters between the policy and value function, we use a loss function that combines the policy surrogate and a value function error term. This objective is further augmented by adding an entropy bonus to ensure sufficient exploration, which is the third term in the above equation.

TABLE I
PPO Hyperparameters

Hyperparameters	**Value**
GAE Lambda	0.95
GAE Gamma	0.9
PPO Epsilon	0.2
PPO Memory Size	2048
Optimizer	Adam
Actor Network Activation Function	Tanh
Critic Network Activation Function	Tanh

Few important inferences that can be drawn from the PPO equation are:

- It is a policy optimization algorithm, that is, in each step there is an update to an existing policy to seek improvement on certain parameters.
- It ensures that the update is not too large, that is the old policy is not too different from the new policy due to clipping of the updated region to a very narrow range.

- Advantage function is the difference between the future discounted sum of rewards on a certain state and action, and the value function of that policy.
- Importance sampling ratio, or the ratio of the probability under the new and old policies respectively, is used for update.
- Epsilon is a hyperparameter that denotes the limit of the range within which the update is allowed.

B. Description of the Algorithm

As shown in the figure 2, the algorithm for proximal policy optimization (PPO) is given. For each iteration, each of N (parallel) actor collect T timesteps of data. Then a surrogated loss on these NT timesteps of data is constructed, and is optimized by a minibatch stochastic gradient descent for K epochs. For a better performance, generally Adam optimizer is used.

What we can observe, is that small batches of observation are used for updation, and then thrown away in order to incorporate a new batch of observation. The updated policy will be clipped to a small region so as to not allow huge updates which might potentially be irrecoverably harmful. In short, PPO behaves exactly like other policy gradient methods in the sense that it also involves the calculation of output probabilities in the forward pass based on various parameters and calculating the gradients to improve those decisions or probabilities in the backward pass. It involves the usage of importance sampling ration. However, it also ensures that the old policy and new policy are at-least at a certain proximity, and very large updates are not allowed.

IV. Deep Deterministic Policy Gradient (DDPG)

DDPG is a model-free, off-policy actor-critic algorithm, combining DPG with DQN. Recall that DQN (Deep Q-Network) stabilizes the learning of Q-function by experience replay and the frozen target network. The original DQN works in discrete space, and DDPG extends it to continuous space with the actor-critic framework while learning a deterministic policy. A key feature of the approach is its simplicity: it requires only a straightforward actor-critic architecture and learning algorithm with very few "moving parts", making it easy to implement and scale to more difficult problems and larger networks.

A. Description of the Algorithm

Inputs to the critic network i.e. state and action are concatenated together which further outputs a value for action in a particular state. We are using a continuous environment that's why we are using tanh activation (output values b/w -1 and 1) and output is the length of action. In DDPG, we have target networks for both actor and critic like in DQN we have a target network. We have also used a replay buffer to store experiences. Further the action is clipped between max and min action value range.
An exploration policy μ' is constructed by adding noise η for further exploration.

TABLE II
DDPG Hyperparameters

Hyperparameters	Value
DDPG actor learning rate	0.001
DDPG critic learning rate	0.002
Gamma	0.99
Tau	0.005
Loss function	MSE
Optimizer	Adam
Fully connected layers	2
Fully connected width	256
Actor network activation function	[Relu, Relu, Tanh]
Critic network activation function	[Relu, Relu]
Batch size	64
Number of epochs	10
Number of episodes	5000

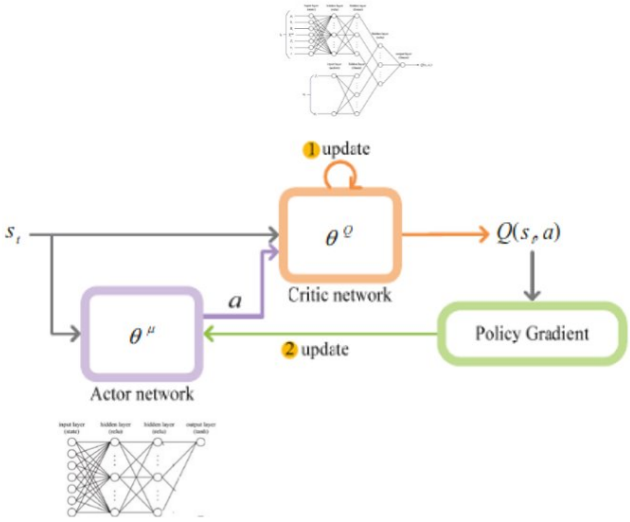

Fig. 5. DDPG Network structure

$$\mu'(s) = \mu_\Theta(s) + \eta$$

In addition, DDPG does soft updates ("conservative policy iteration") on the parameters of both actor and critic, with : $\tau \ll 1 : \Theta' \leftarrow \tau\Theta + (1-\tau)\Theta'$. In this way, the target network values are constrained to change slowly, different from the design in DQN that the target network stays frozen for some period of time.

V. Asynchronous Advantage Actor Critic (A3C)

A3C is a conceptually simple and lightweight framework for deep reinforcement learning that uses asynchronous gradient descent for optimization. The most common Deep Reinforcement Learning algorithms use the concept of a experience replay buffer, which is basically a collection of the environments state, action, reward combinations which are used for training the model. The model learns from the experience replays in order to understand the environment behaviour. However, having an experience replay has several drawbacks. First, it uses more memory and computation per real interaction. Second, it requires

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. 6. DDPG Algorithm

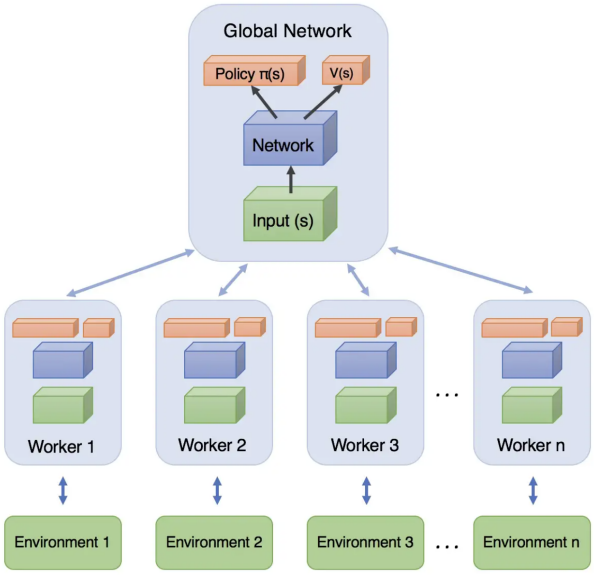

Fig. 7. A3C Network

off-policy learning algorithms that can be updated from data generated by an older policy. A3C claims to overcome these drawbacks either completely, or at the very least partially, while providing similar output rewards.

The principle idea behind this asynchronous algorithm is to implement multiple agents in parallel on multiple instances of the same environment. These agents are independent of each other and keep interacting with the environment in order to learn the policy of the environment. These agents in the future update a global network in an asynchronous fashion. This parallelism avoids any correlation between the agents' data into a more stationary process, since at a given time-step the parallel agents will be interacting with a different state each. This idea of training multiple agents in parallel can be applied to multiple Reinforcement Learning algorithms to make them

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  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}
    $t \leftarrow t+1$
    $T \leftarrow T+1$
  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': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i;\theta'_v))^2/\partial\theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Fig. 8. A3C Algorithm

robust and effective in their implementations. The actor-critic model studied here is an on-policy policy search method.

A. Advantage of A3C algorithm

The main idea behind A3C can be broken down into two parts: First, instead of using separate machines or high-performance GPUs, we can use multiple CPU threads on a single machine. Second, we make the observation that multiple learners running in parallel are likely to be exploring different parts of the environment. Moreover, it is possible to use different exploration policies in each actor-learner to maximize this diversity. By using different exploration policies on multiple instances of the environment, we avoid correlation between the actor-learners. This employs a higher stabilizing effect than an experience replay in the DQN training algorithm.

Key outcomes of the asynchronous actor-critic model:

1. Reduction in training time that is roughly linear to the number of parallel actor-learners

2. We no longer need to rely on experience replay for stabilizing learning process. We can use on-policy reinforcement learning methods to train neural networks in a stable way.

B. Description of Algorithm

A3C maintains a policy $\pi(a_t|s_t;\theta)$ and an estimate of the value function $V(s_t;\theta_v)$. The actor-critic operates in the forward view and uses a mix of n-step returns to update both the policy and the value function. The policy and the value function are updated after every t_{max} actions or when a terminal state is reached.

The update from the algorithm is:

$$\nabla_{\theta'} \log \pi(a_t|s_t;\theta') A(s_t, a_t; \theta, \theta_v)$$

where $A(s_t, a_t; \theta, \theta_v)$ is an estimate of the advantage function given by $\sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k};\theta_v) - V(s_t;\theta_v)$, where k can vary from state to state and is upper-bounded by t_{max}.

TABLE III
A3C HYPERPARAMETERS

Hyperparameters	Tested Values
Number of Episodes	5000 and 10000
A3C Gamma (Generalization)	0.99
Global Update Frequency	500
RMSProp Epsilon	0.00001
Number of workers	8 and 16
Actor Coefficient	1.0
Critic Coefficient	0.5
Entropy Coefficient	0.01
Spawned Vehicles	3
Critic Loss function	MSE
Optimizer	RMSProp
Fully connected layers	3
Fully connected width	256
Actor network activation function	[Relu, Tanh]
Critic network activation function	[Relu, Softmax]

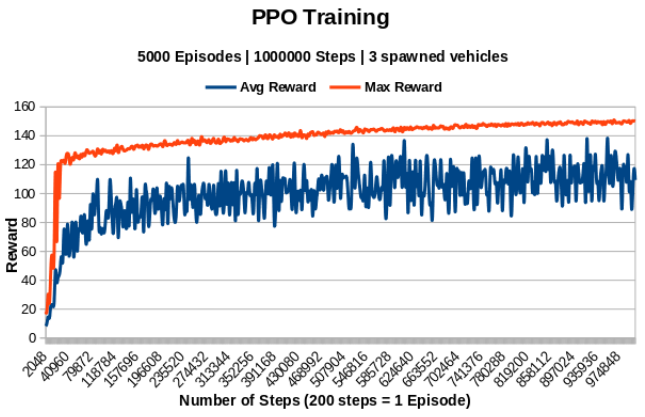

Fig. 9. PPO Training

We typically use a convolutional neural network that has one softmax output for the policy and one linear output for the value function, with all non-output layers shared. It is also found that adding the entropy of the policy π to the objective function improved exploration by discouraging premature convergence to suboptimal deterministic policies.

VI. TRAINING RESULTS

A. A3C Training Results

Based on our implementation, we plotted the results for each algorithm to show the average and maximum rewards against the number of steps or episodes. These results can be seen in the figures; Figure 9 shows PPO Training results, Figure 10 shows DDPG Training results and Figure 11 shows A3C training results.

PPO results show an increase in the average rewards as expected. DDPG shows an increase in average rewards but with a lot of fluctuations over time as compared to the PPO rewards. However, A3C did not train well in our implementation for 5000 episodes with 3 spawned vehicles.

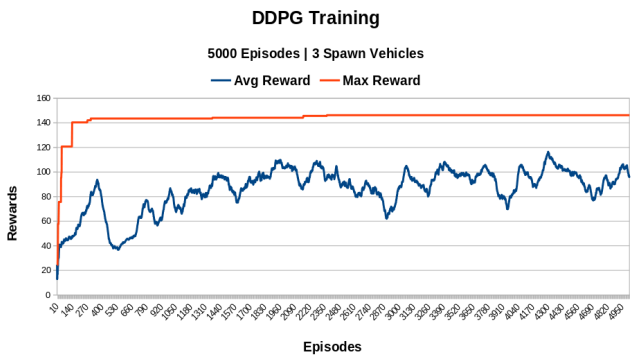

Fig. 10. DDPG Training

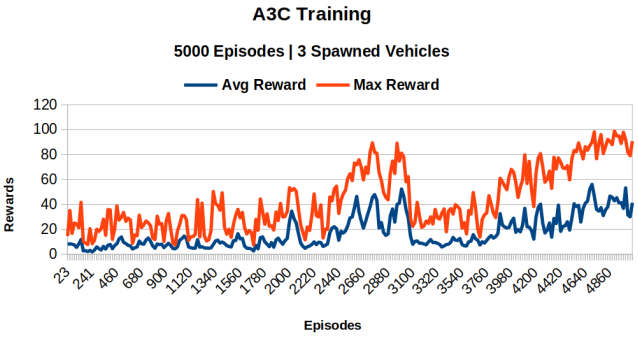

Fig. 11. A3C Training

VII. TESTING RESULTS

The following table shows testing results on experiments conducted by spawning 0, 3 and 8 random vehicles.

TABLE IV
AVERAGE REWARDS OF DIFFERENT AGENTS TESTED WITH 0(NO DENSE), 3(LIGHTLY DENSE), AND 8(HEAVILY DENSE) SPAWN AGENTS

Agents	0	3	8
PPO	152	126	72
DDPG	142	38	38
A3C	92	92	29

VIII. CONCLUSION

We see that deep reinforcement learning algorithms can be implemented in order to train agents to perform tasks almost as well as humans. This project shows one such application in the autonomous vehicle industry, where we can have Reinforcement Learning agents assist drivers in lane keeping and avoiding obstacles. This project compares basic Reinforcement Learning algorithms in order to demonstrate the same. Based on our analysis of the 3 algorithms: PPO, DDPG and A3C, we observe the following key conclusions.

- From the testing results it is clear that PPO performs better as compared to DDPG and A3C algorithm.
- Average reward decreases with increase in the number of spawned agents. This implies that the agents trained

on environments with 3 spawned vehicles does not generalize well for increased number of agents.

- The training time for PPO and DDPG for 5000 episodes is almost same whereas the training time needed for A3C was comparatively lesser and the reason could be as A3C doesn't use GPUs and utilizes multiple CPU cores depending upon the number of workers. This means the training process in A3C is a parallel process which updates the global network through multiple actor-critic agents simultaneously and asynchronously.

REFERENCES

[1] Schulman, John and Wolski, Filip and Dhariwal, Prafulla and Radford, Alec and Klimov, Oleg (2017) arXiv - Proximal Policy Optimization Algorithms.

[2] Lillicrap, Timothy P. and Hunt, Jonathan J. and Pritzel, Alexander and Heess, Nicolas and Erez, Tom and Tassa, Yuval and Silver, David and Wierstra, Daan (2015) arXiv - Continuous control with deep reinforcement learning

[3] Mnih, Volodymyr and Badia, Adri'a Puigdom'enech and Mirza, Mehdi and Graves, Alex and Lillicrap, Timothy P. and Harley, Tim and Silver, David and Kavukcuoglu, Koray (2016) arXiv – Asynchronous Methods for Deep Reinforcement Learning

[4] Edouard Leurent, Highway-env – https://github.com/elurent/highway-env

[5] Lilianweng Blog - https://lilianweng.github.io/posts/2018-04-08-policy-gradient/ddpg

[6] https://towardsdatascience.com/deep-deterministic-and-twin-delayed-deep-deterministic-policy-gradient-with-tensorflow-2-x-43517b0e0185

[7] Shagun Maheshwari (Medium) - Implementing the A3C Algorithm to train an Agent to play Breakout!