Distributed Approach for implementation of A3C on TORCS

Ameya Wagh
Robotics Engineering,
WPI, MA 01609
aywagh@wpi.edu

Shakthi Duraimurugan
Robotics Engineering,
WPI, MA 01609
sduraimurugan@wpi.edu

Sanket Gujar
Computer Science,
WPI, MA 01609
srgujar@wpi.edu

***Abstract*—The performance of a reinforcement learning agent depends on its exploration by interacting with the environment. A popular approach to increase the exploration is to implement Asynchronous advantage actor critic algorithm, but it requires a multi-core high-performance CPU which can multi-thread environment processes. We propose a simple scalable distributed framework to implement the Asynchronous advantage actor-critic reinforcement algorithm on multiple single or dual core systems. This framework was experimented on the open source car racing simulator-TORCS. Based on the results obtained, we conclude that a distributed approach with a high number of agents gives more exploration to the agent and reduces the training time than a single agent running on a single system. Furthermore, the proposed framework can be experimented with other deep reinforcement learning algorithms.**

***Keywords*—distributed system; asynchronous advantage actor critic; TORCS; Reinforcement Learning**

I. Introduction

With ever-increasing computational power, we are recently seeing a rising trend towards deep reinforcement learning for complex tasks. The deep neural networks are able to provide effective models that enable reinforcement learning. A popular method among them has been the Deep Q-Network. They have been highly successful in the classic ATARI 2600 games, even better than humans in some cases.

However, a big limitation of Deep Q-networks is that the outputs/actions are discrete. While in environments such as car racing, the action space is continuous. If we try discretizing it, there arises another problem of high dimensionality. Also, Deep Q-networks work on the idea that updates can be decor-related by randomly sampling stored data from different time steps. This requires a lot of memory and is computationally expensive.

To solve these drawbacks, Googles DeepMind came up with an algorithm called Asynchronous advantage actor-critic [1]. Its basic idea is to asynchronously run multiple actor-critic agents in parallel on multiple instances of the environment. The algorithm that we propose in this paper uses a distributed framework of actor-critic rather than multithreading it on a single system. This means that some old computers running on outdated CPUs can effectively replace the need for a GPU or a multi-core CPU. We implement the algorithm on an open source car racing simulator TORCS. The advantages of this method are:

- It can be used for continuous action spaces.
- It is not computationally expensive and does not require huge memory as it has a smaller experience replay.
- It does not require high-end GPUs and CPUs as the framework is distributed on many systems.
- Highly scalable to any number of systems.
- Increased redundancy as even if one system crashes, the others would run and continue to update the policy

In the following sections, the paper discusses the related work done in Asynchronous reinforcement learning and TORCS simulator, then gives a background of reinforcement learning, followed by the methodology used and the obtained results.

II. Related Work

The application of computational intelligence to car racing games has been investigated in several works. In 1998, Pyeatt and Howe [2] applied reinforcement learning to learn racing behaviors in RARS, an open source racing simulator, precursor of The Open Racing Car Simulator (TORCS) used in this work. Benoit Chaperot and Colin Faye [3] investigated methods to improve the back-propagation algorithm to have the computer controlled bikes performing as well or better than a human player. D. Perez, G. Recio, Y. Saez, and P. Isasi [4] developed a controller with fuzzy rules and fuzzy sets for input and output, which were evolved using a genetic algorithm in order to optimize lap times, damage taken and out of track time.

Asynchronous methods for reinforcement learning has been gaining popularity recently. Gorila framework by (Nair et al., 2015) [5] uses massively distributed architecture for deep reinforcement learning in which agents are distributed. It uses 4 main components, parallel actors, learners, distributed Neural network and distributed experience buffer. They implemented the Deep-Q network algorithm (Mnih et al., 2013) [6] in their distributed architecture on ATARI 2600 games. DistBelief (Dean et al) [7] paper talks about a software framework to scale learning models over large number of clusters. They developed 2 algorithms, Downpour SGD and Sandblaster. Distributed Deep Q-Learning (Chavez et al., 2015) [8] they adapted the DistBelief Framework to implement agent which learns control policies from high-dimensional inputs. (Koutnik et al.,2014) [9] used neuro evolutionary algorithm in which they evolved recurrent neural network controller to drive a car in TORCS game using a compressed. (Tomassini 1999) [10] uses parallel approach by distributing it on multiple machines to solve hard problems.

III. REINFORCEMENT LEARNING BACKGROUND

A. Off-Policy vs On-policy

Reinforcement learning algorithms can be generally characterized as off-policy where they employ a separate target behavior policy that is independent of action policy improved upon. The benefit of this separation is that the target behavior policy will be more stable by sampling all actions, whereas the action estimation policy can be greedy, giving more exposure to the agent. Q learning is an off-policy algorithm as it updates the Q values without making any assumptions about the actual policy followed. Whereas, On-policy directly uses the policy that is being estimated to sample trajectories during training. [11]

B. Model Free Algorithms

Model-free algorithms are used where there are highdimensional state and action spaces, where the transition matrix is incredibly expensive in space and time to compute. Modelfree algorithms makes no effort in learning the dynamics that governs how an agent interacts with the environment, it directly estimates the optimal policy or value function by policy iterations or value iterations, but also model-free algorithms need a large number of training examples for proper approximations.

C. Experience Replay

During all plays the experience (s, a, r, s) are stored in replay memory. So, while training the network random mini batches from the replay memory are used instead of the most recent transactions. This breaks the similarity of subsequent training samples, which might drive the network in local minimum. [12]

D. Actor-Critic model

Actor-critic (AC) implements generalized policy iteration alternating between a policy evaluation and a policy improvement steps.

A hybrid model which combines policy gradient (known as actor) and value function (known as critic) together. The actor produces the action (a) given the current state(s), the actor and critic are connected in the action, which is a part of the critics input, which calculates the Q-value.

The critic's main purpose is to criticize and train the actor and uses temporal difference to determine whether an action was worse or better than expected. While learning this difference is back-propagated through the critic and then through the actor.

E. A3C (Asynchronous Advantage actor-critic)

1) Asynchronous: A3C utilizes multiple agents with multiple environment interaction to learn more efficiently. A3C had a global network and multiple worker agent which have their own set of network parameter. Each agent interacts with its own environment simultaneously, so the experience of one agent is independent of others, making the overall experience available for training become more diverse.

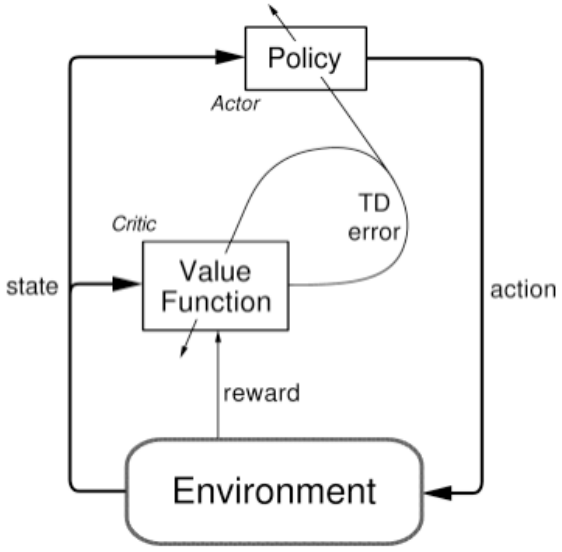

Fig. 1: Actor Critic Model : AC combines policy gradient (known as actor) and value function (known as critic), where critic's main purpose is to criticize and train the actor and using temporal difference to determine whether an action was worse or better than expected. While learning, this temporal difference is back-propagated through the critic and then through the actor.

2) Advantage: The advantage helps the agent to determine how much its action turned out to better or worse than expected, rather than just using the discounted returns which only tells if the actions were good or bad. We used discounted reward as an estimate of Q value for a state and action as we didnt determine the Q value directly in A3C

$$\text{Discounted reward} : R = \gamma(r) \tag{1}$$

$$\text{Advantage} : A = Q(s,a) - V(s) \tag{2}$$

$$\text{Advantage Estimate} : A = R - V(s) \tag{3}$$

F. Gym TORCS

Gym-TORCS [13] is a reinforcement learning environment with Open-AI gym interface for a racing car game Torcs [14]. It is an open-source realistic car racing simulator used as RL benchmark. The current implication is for only-the single-track race in practice mode.

IV. METHODOLOGY

A. Framework

The framework consists of an HTTP server and multiple workers running on different machines connected over internet. Every worker is an HTTP client running an agent with Asynchronous Advantage Actor critic algorithm referred as worker model in the above figure. The server has a target model which is updated by all workers asynchronously i.e. the start and duration of each episode for every worker is

Fig. 2: Front facing camera view from a car of Torcs practice arena, The Dial on the bottom right represent the magnitude of actions taken by the agent.

Algorithm 1 Pseudo code for implementation of Server

Initialize target model
Load target model with stored weights $target_{weights}$
function ONEVENT(/Upload)
    $target_{weights} \leftarrow \tau * worker_{weights} + (1-\tau) \times target_{weights}$
function ONEVENT(/Download)
    $http_response \leftarrow target_{weights}$
function ONEVENT(/)
    $http_response \leftarrow render\ webpage$
function EVENT HANDLER
    $handler \leftarrow register(/Upload, /Download, /)$

architecture.pdf

Fig. 3: Architecture of Framework: The Worker models are interacting with with their own copy of Torcs environment. The worker agent pushes the network weights to the server when an episodes end and pulls network weights from the server before starting a new episode.

independent. All the worker models are trained locally and simultaneously during the game play using Replay buffer and send trained weights to the server of end of every episode or any abrupt program crash. This framework gives liberty for every worker to have different parameters which increase exploration and experience due to which local minima can be avoided

At the start of every Episode, the worker pulls the target model weights from the server and starts to train on it. At the end of every episode it pushes the trained worker weights to the server as well as saves a local copy. This local copy is used when there is no communication possible between the server and the worker. In these situations, the workers would run independently till a connection is re-established. On every event when worker uploads the model weights the server updates the target value with a scaling factor τ

$$targetweights \leftarrow \tau * Workerweights + (1-\tau) * targetweights \tag{4}$$

When a new worker joins the network, it gets the updated weights, thus it starts to explore from the current knowledge and experience of the policy than starting from scratch.

Algorithm 2 Pseudo code for implementation of Worker

Initialize worker model
function LOADMODEL **return** *model*
    if pingServer() == true **then**
        $worker_{weights} \leftarrow pullFromServer()$
    else
        $worker_{weights} \leftarrow load\ previously\ saved\ model$
for episode in Episodes **do**
    $agent \leftarrow loadModel()$
    $TorcsEnv.reset()$
    for steo in episode **do**
        $Obs, Reward, doneFlag \leftarrow TorcsEnv.step(action)$
        $action \leftarrow agent.act(Obs, Reward)$
        if doneFlag == true **then**
            $localDisk \leftarrow saveModel()$
            $server \leftarrow pushModel()$
            $break$

B. Agent

The gym-TORCS runs as a client to a customized version of TORCS server and communicates over UDP. Every program communicates to the server on a static port. In this paper multiple TORCS games were running on different machines on port 3101. The TORCS game provides several sensors as observation and different controls as action.

1) Observation space: The observation space of gym-TORCS contains vision input as well as many sensors which can be seen in Table 1 of Appendix.

In the scope of this experiment, only two variables are used namely, angle i.e. angle made by the heading of the car to the center of the track and trackPos i.e. the position of the car from the track axis. The angle varies from [-,] and the trackPos ranges from [-1,1] -1 being the car is on extreme right edge.

2) Action-Space: The TORCS provides a no of variables to control the car using its UDP client which can be seen in Table 2 of Appendix.

The action space consists of just one variable which is the steering angle which lies in the range -1 to 1. The acceleration and breaking are automatic and are a function of steering angle. i.e. when the car turns it decelerates and when it is going straight it accelerates accordingly.

3) Design of reward function: We initially started with a simple reward function which was the velocity of the car along the track directions Vx, time for which the agent stays on the

track t and negative rewards for collisions.

$$R_t = V_x + t \tag{5}$$

We observed that the agents get stuck in local minimums by hitting accelerator and hit the edges of the tracks. This was dealt by [15] by maximize longitudinal velocity $V_x * \cos(\theta)$, minimize transverse velocity $V_x * \sin(\theta)$ and maintain the agent in the center of the track.

$$R_t = V_x \cos(\theta) - V_x \sin(\theta) - V_x trackPos \tag{6}$$

4) Randomization process: To avoid local minima and to increase exploration a random action value is given from a normally distributed action space. To control the frequency of random actions another random number is selected from range (0,1) and compared against exploration factor . If the value is greater than a random action is taken from normally distributed action space else the prediction of actor is used as action.

C. Web Interface

The interface is used to monitor the process displayed on a webpage. Fig 4. shows that it can display the Workers attached to the server, Maximum reward gained, no of episodes elapsed and server uptime with configuration messages and logs.

The webpage also plots a real-time graph of the global rewards vs global no of episodes elapsed shown in Fig 5. Global rewards indicate the rewards by all the workers together and global episodes are episodes elapsed by all the workers together.

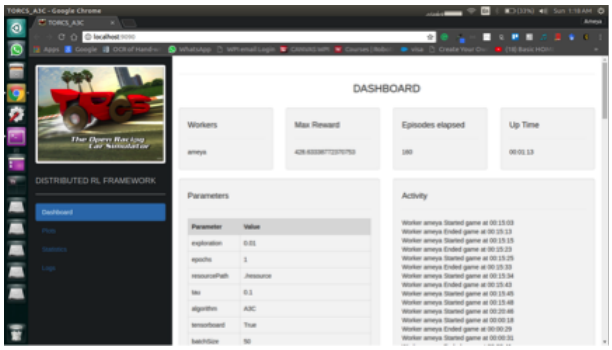

Fig. 4: Webpage Dashboard: The workers, Max Reward, Episodes, Up Time, Parameters of the network and Activity informations are displayed to the user.

D. Evaluation Protocol

We would evaluate the rewards accumulated by a two agent system vs. a single agent system, and assess the factor by which rewards accumulated increases from one system to the other. We would also check if this factor corresponds to the results obtained in the implementation of A3C in Table 2 of paper [1].

Fig. 5: Webpage Dashboard: Plot of Global Reward vs Global Episodes

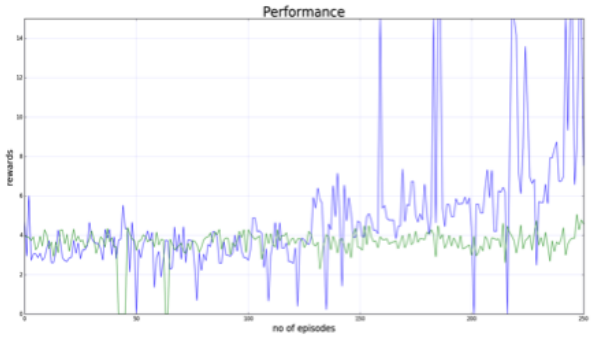

Fig. 6: Plot of rewards vs epsiodes for single-agent(green) and two-agent systems(blue)

V. RESULTS

Given below are the graphs of the rewards of the algorithm vs elapsed episodes performed with one agent and two simultaneous agents. It was performed on dual core Intel i7 CPUs. It can be observed that in a span of 250 episodes the experiment performed with two simultaneous agents shows significant improvement.

The practice track in the Torcs game over which the agents were trained consists of steep turns and straight roads. The game starts with a turn followed by a road. Thus, for an untrained agent it becomes very difficult to learn to maneuver a turn right at the beginning. As the time elapses, the agent progresses to make a turn and drive straight and will crash only at the next turn.

It can be seen in Fig 6. that there are significant peaks after 150 episodes in the two-agent system. These peaks correspond to successful maneuvers of the first hard turn. As the agent progresses to maneuver the car it sometimes barely makes the turn aided by Random explorations. The frequency of these peaks increases over time finally learning to correctly take a turn. The single-agent experiment was not able to reach that level in 250 episodes.

The downward peaks in the graph signify the random actions taken in the exploration stem causing the car to crash at the beginning.

It can also be observed that both the experiments perform

similarly at the start of the training but the experiment with two-agents learns faster as it has two independent agents to explore the same observation space signifying the effect of parallelization.

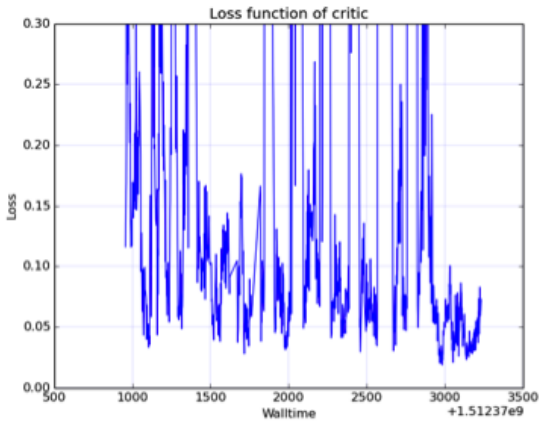

Fig. 7: Plot of Loss function of critic network (Value function) over training time

The critic is a regression mode which learns to gives value of every observation state. The figure 7 show the trend of decrease in loss over time of the critic mode stating the improvement in the performance of the critic to evaluate a given observation.

From Fig 6. we can observe that the reward gained by two-agent system in 250 episodes increased by approximately a factor of 2 which agrees with the result obtained in Table 2 of paper[1].

VI. CONCLUSION AND FUTURE SCOPE

We presented a distributed approach to perform A3C on open source car racing environment TORCS and showed that we were able to train neural network agents to drive the car in the mentioned environment by gaining maximum rewards. We performed two experiments on the environment using the distributed approach on single-agent and two-agent with dual core Intel CPUs, From the above results we conclude that multiple asynchronous independent agent running parallel on distributed systems learn faster. The two workers had different parameters which increased the exploration.

This distributed approach does not limit on no of workers and thus can be scaled largely. As the no of workers increase the exploration of the environment increases which makes the agent less likely to get stuck in local minimums.

The downside of these systems is latency due to communications and IO operations. This approach is suitable for clusters with low performance single or dual core CPUs, but it can be replaced by a single high performance multithreaded system.

We can also combine other reinforcement learning methodologies in this distributed approach to make them asynchronous.

VII. ACKNOWLEDGMENT

We would like to thank Prof. Dmitry Korkin for his valuable guidance and suggestions.

REFERENCES

[1] Volodymyr Mnih, Adri'a Puigdom'enech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu, "Asynchronous methods for deep reinforcement learning", In Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016, pages 19281937, 2016.

[2] L. D. Pyeatt and A. E. Howe, Learning to race: Experiments with a simulated race car, in Proceedings of the Eleventh International Florida Artificial Intelligence Research Society Conference. AAAI Press, 1998, pp. 357361

[3] B. Chaperot and C. Fyfe, Improving artificial intelligence in a motocross game, in IEEE Symposium on Computational Intelligence and Games, 2006.

[4] D. Perez, G. Recio, Y. Saez, and P. Isasi, Evolving a fuzzy controller for a car racing competition, in Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on, Sept. 2009, pp. 263 270

[5] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. D. Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver, Massively Parallel Methods for Deep Reinforcement Learning, [1507.04296] Massively Parallel Methods for Deep Reinforcement Learning, 16-Jul-2015. [Online]. Available: https://arxiv.org/abs/1507.04296. [Accessed: 03-Dec-2017].

[6] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. [online] Arxiv.org. Available at: https://arxiv.org/abs/1312.5602v1 [Accessed 4 Dec. 2017].

[7] J. Dean, G.S. Corrado, R. Monga, K. Chen, M. Devin, Q.V. Le, M.Z. Mao, M.A. Ranzato, A. Senior, P. Tucker, K. Yang, A. Y. Ng., Large Scale Distributed Deep Networks, NIPS, 2012

[8] Ong, H., Chavez, K. and Hong, A. (2015). Distributed Deep Q-Learning. [online] Arxiv.org. Available at: https://arxiv.org/abs/1508.04186 [Accessed 4 Dec. 2017].

[9] Koutnk, Jan, Schmidhuber, Jrgen, and Gomez, Faustino. Evolving deep unsupervised convolutional networks for vision-based reinforcement learning. In Proceedings of the 2014 conference on Genetic and evolutionary com- putation, pp. 541548. ACM, 2014.

[10] Tomassini, Marco. Parallel and distributed evolutionary al- gorithms: A review. Technical report, 1999.

[11] Emami,P.(2017) http://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html [Blog].

[12] Matiisen,T.(2017).https://www.intelnervana.com/demystifying-deep-reinforcement-learning/ [Blog].

[13] Gym TORCS.https://github.com/ugo-nama-kun/gym_torcs [Online]

[14] The open racing car simultaor website, http://torcs.sourceforge.net/ [Online]

[15] Lau, B. (2016). Using Keras and Deep Deterministic Policy Gradient to play TORCS.[Blog] Available at:https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html [Accessed 4 Dec. 2017].

VIII. APPENDIX

TABLE I: Different sensor outputs from TORCS environment

Name	Range(unit)	Description
angle	[-,+] (rad)	Angle between the car direction and the direction of the track axis.
speedX	(,+) (km/h)	Speed of the car along the longitudinal axis of the car.
speedY	(,+) (km/h)	Speed of the car along the transverse axis of the car.
speedZ	(,+) (km/h)	Speed of the car along the Z axis of the car.
trackPos	(,+)	Distance between the car and the track axis. The value is normalized w.r.t to the track width: it is 0 when car is on the axis, -1 when the car is on the right edge of the track and +1 when it is on the left edge of the car. Values greater than 1 or smaller than -1 means that the car is outside of the track.

TABLE II: Different controls for the TORCS environment

Name	Range(unit)	Description
accel	[0,1]	Virtual gas pedal (0 means no gas, 1 full gas).
brake	[0,1]	Virtual brake pedal (0 means no brake, 1 full brake).
steering	[-1,1]	Steering value: -1 and +1 means respectively full right and left, that corresponds to an angle of 0.366519 rad.