International Journal of Computer Applications Technology and Research
Volume 11–Issue 07, 284-290, 2022, ISSN:-2319–8656
DOI:10.7753/IJCATR1107.1005

Implementation and Evaluation of Advantage Actor-Critic Algorithm on a Desktop Computer with a Multi-Core CPU

Fredy Martínez
Universidad Distrital Francisco José de Caldas
Bogotá D.C., Colombia

Angélica Rendón
Universidad Distrital Francisco José de Caldas
Bogotá D.C., Colombia

Abstract: In this paper, the implementation and evaluation of the Advantage Actor-Critic (A2C) algorithm, one of the most important Deep Reinforcement Learning schemes, is performed. The objective is to determine the behavior of the algorithm on a desktop computer with a multi-core CPU, establishing its behavior, performance, and resource consumption for embedded applications. This algorithm makes use of multiple agents on parallel instances of the environment so that each agent adds knowledge to the system, which is weighted by a value of Advantage that evaluates its interaction in the environment. This assessment is performed on OpenAI's CartPole-v0 playground, so the results are comparable and easily reproducible. The results show a high performance of the algorithm for different instances with fixed-length segments of experience, which allows us to think of successful use on more resource-constrained hardware platforms.

Keywords: A2C; agent; cartpole-v0; environment; optimal policy; reinforcement learning; value function

1. INTRODUCTION

Reinforcement learning (RL) is a Machine learning strategy inspired by the behaviorist psychology of John B. Watson, under which the actions of agents are determined by their interaction with the environment, leaving aside consciousness and introspection [1]. Under this idea, the agent makes decisions to maximize some reward or reward function throughout their interaction [2, 3]. Given its simplicity of the concept, and its high performance in a multitude of tasks, the strategy has been successfully used in many disciplines such as control, game theory, search problems, optimization, and even robotics [4–6]. Although in much of the research related to RL the problem tends to focus on the search for optimal solutions [7], its usefulness as a learning strategy and in tasks related to generalization and approximation has been widely documented and appreciated [8, 9]. In addition, great similarities have been observed with the Markov Decision Process (MDP) given how the agent is related to the environment, which is an important tool of Machine learning, but with the possibility of being used in highly complex problems since the RL does not require explicitly defining the MDP relationships [10]. Another important feature of RL is that, unlike supervised learning, RL does not require Input/Output training pairs, but takes the information for the model from exploration and interaction in the environment [11].

RL models consist of a set of states of the environment, a set of actions, a set of rules for switching between states, rules for evaluating the reward associated with state change, and rules for interpreting the agent's observations. In this sense, a system built to be trained as an RL model can also be described by the nomenclature of a hybrid system [12] or as a reactive system described by Linear Temporal Logic (LTL) [13, 14]. It is precisely this type of structure that allows the use of deep neural networks as the agent's learning architecture to structure what has been called Deep Reinforcement Learning (DRL) [15]. In this framework, the agent's decisions tune a neural model that defines the behavioral policy, which is reflected in the agent's action on the environment (Fig. 1).

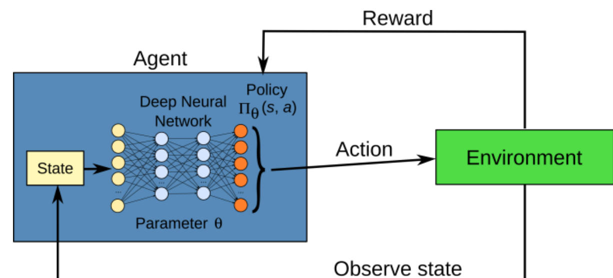

Figure 1. Deep Reinforcement Learning scheme.

One of the most recent DRL-type algorithms, and among the most influential, is the Asynchronous Advantage Actor-Critic (A3C) [16]. The algorithm is called asynchronous because unlike other DRL algorithms with a single agent in a single environment, A3C uses multiple agents each in its environment with its parameters [17, 18]. Each agent interacts in its environment asynchronously with the other agents, learning from its interaction. As each agent gains knowledge, this knowledge contributes to the knowledge of the overall algorithm. This scheme is similar to the experience gained individually by each person but contributes to the joint development of projects. An alternative implementation of the algorithm allows each actor to finish its learning segment before performing an update, which achieves synchronization of the agents, thus forming a better-performing algorithm called Advantage Actor-Critic (A2C) [19, 20]. The Actor-Critic role names are assigned since the algorithm uses a Value function $V(s)$ (Critic) to update the Optimal Policy function $\Pi(s)$ (Actor). The Advantage designation is given since the agent receives a value of Advantage instead of a reward, which improves the learning process.

In this paper, the implementation and evaluation of the A2C algorithm are performed. To generate knowledge, multiple agents are run synchronously in parallel, each in its

environment (all identical instances), letting each one finish its interaction process before starting a new epoch of the system adjustment [21]. At any instant of the tuning process, each agent experiences a different state, which allows the algorithm to decorrelate the data of each agent. The algorithm is run on a desktop computer with a multi-core CPU to determine its actual performance without the use of GPUs. The algorithm is evaluated with OpenAI's CartPole-v0 playground, in which the agent is a cart controlled by two possible actions that cause left and right movements [22].

2. METHOD

In our implementation of the Advantage Actor-Critic algorithm, we consider the traditional RL approach in which an agent interacts with an environment $\boldsymbol{E}$ over time, observed in a finite number of discrete steps (Fig. 1). At each of these time steps, the agent observes a state s_t in the environment, to which it responds with action a_t selected from some set of possible actions $\boldsymbol{A}$, and in coherence with a behavioral policy π. The action value $Q^{\pi}(s,a)$ (Eq. 1) corresponds to the expected return for the selected action a in state s and according to policy π.

$$Q^{\pi}(s,a) = \boldsymbol{E}[R_t|s_t = s,a] \qquad (1)$$

Under this context, Π (set containing all π) is a mapping from states to actions. Upon executing this action, the agent receives a reward r_t, and the state of the system evolves to a new observable state $s_{(t+1)}$. This process continues for each time step until a final state is reached, evolving in such a way that the agent at each step maximizes the reward received. In the end, the cumulative return is given by Eq. 2 for a discount factor $\gamma \in (0,1]$.

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \qquad (2)$$

In value-based RL strategies, the function value is approximated utilizing some suitable model, e.g., a neural network. In this case, the approximator takes the following form (Eq. 3):

$$Q^{\pi}(s,a) \approx Q(s,a;\theta) \qquad (3)$$

This corresponds to an action-value function approximator with θ parameters, where the value of such parameters is iteratively updated by various RL algorithms, as in the case of Q-learning, again maximizing the reward it receives at each transition. As an alternative to value-based methods there are policy-based methods, in which the policy is parameterized in the form (Eq. 4):

$$\pi(a|s;\theta) \qquad (4)$$

Here again, the θ parameters are adjusted, but in this case through an ascending gradient over the environment considering the cumulative return.

The variance of the estimate can be reduced by subtracting the learned function of the state $b_t(s_t)$ from the cumulative return value R_t. This parameter R_t-b_t becomes an estimator of the advantage of the action a_t in state s_t. It should be remembered that R_t is an estimate of Q^{π}, and that b_t is an estimate of $V^{\pi}(s_t)$. This structure is known as actor-critic architecture (policy π is the actor and b_t is the critic).

Asynchronous Advantage Actor-Critic (A3C) is a robust, simple, and high-performance Deep RL algorithm compared to other Deep RL schemes, particularly in tasks with complex state and action spaces (Fig. 2). It is called asynchronous because the algorithm uses multiple agents, each in its environment, which are trained in parallel, contributing their experience (knowledge gained) individually regardless of the progress of the other agents (asynchronous update). The advantage is the metric used to evaluate the actions of each agent, and the Actor-Critic model is used for decision making (actor) and the evaluation of how good the action was (critic).

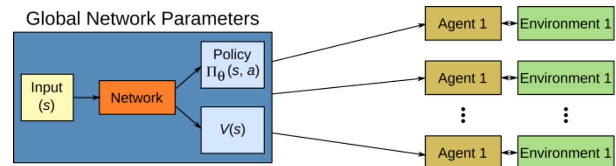

Figure 2. Asynchronous Advantage Actor-Critic (A3C).

Advantage Actor-Critic (A2C) is a variant of A3C, with similar performance, in which it waits for each agent to finish its experience segment in the environment before performing the update, which makes it simpler than A3C, and suitable for running on CPU-only machines (no GPU). A coordinator is included in the overall scheme, which activates each agent in sequence, reducing the computational cost of the algorithm (Fig. 3).

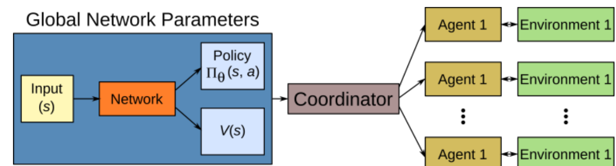

Figure 3. Synchronous Advantage Actor-Critic (A2C).

Our implementation of the A2C algorithm was developed in Google Colab because of its accessibility without prior configuration, interactivity, the possibility of using GPU, and ease of sharing content. A Google account is sufficient to access this tool. A recommended first step is to connect the Google Drive service to the Google Colab virtual computer, which can be done as illustrated in Fig. 4.

```
1 from google.colab import drive
2 drive.mount('/content/drive/')
```

Figure 4. Connecting Google Drive to Google Colab.

After account verification, the Drive storage service is connected to the virtual machine. The next step is to import the necessary libraries. In addition to numerical manipulation and visualization, the Torch tools, an open-source machine learning library used to implement the learning model, must be imported (Fig. 5). Also in this part, it is convenient to configure CUDA in case you have and want to use this capability in the hardware.

```
1  import sys
2  import math
3  import random
4  import gym
5  import numpy as np
6
7  import torch
8  import torch.nn as nn
9  import torch.optim as optim
10 import torch.nn.functional as F
11 from torch.distributions import Categorical
12 from IPython.display import clear_output
13 import matplotlib.pyplot as plt
14 %matplotlib inline
15
16 use_cuda = torch.cuda.is_available()
17 device   = torch.device("cuda" if use_cuda else "cpu")
```

Figure 5. Import of Torch and other libraries.

The environment used is CartPole-v0, OpenAI's gym was also imported with the other libraries. It is a simple playground

widely used to train and test RL algorithms, which in turn makes it a platform that guarantees performance comparison between different strategies and implementations. In this environment, the agent is a cart balancing a vertical bar (inverted pendulum), which is controlled by two possible actions that force it to move to the right or the left. A reward of +1 is assigned to each step that manages to keep the bar in its vertical position since the control objective is to prevent it from falling to either end. The goal of our model is to maximize the total reward throughout the process, which would guarantee that the bar does not fall, and therefore that the problem has been solved. Before creating the environment, it is necessary to install some libraries to the virtual machine to perform the simulations (Fig. 6).

```
!apt-get install python-opengl -y
!apt install xvfb -y
!pip install pyvirtualdisplay
!pip install piglet
```

Figure 6. Installation requirements for the environment.

After fulfilling these requirements, the next step is to create the environments. In our case, we are creating 20 environments (Fig. 7).

```
1  from pyvirtualdisplay import Display
2  Display().start()
3
4  num_envs = 20
5  env_name = "CartPole-v0"
6
7  def make_env():
8      def _thunk():
9          env = gym.make(env_name)
10         return env
11
12     return _thunk
13
14 envs = [make_env() for i in range(num_envs)]
15 envs = SubprocVecEnv(envs)
16
17 env = gym.make(env_name)
```

Figure 7. Creation of environments.

The other important element of this Deep RL model is the deep neural network. For it, a method was created capable of constructing the network from the needs, stacking layers with the Sequential model of the required size. The activation function used in the hidden layers is ReLU and in the output layer Softmax (Fig. 8).

```
1  class ActorCritic(nn.Module):
2      def __init__(self, num_inputs, num_outputs, hidden_size, std=0.
                                                                0):
3          super(ActorCritic, self).__init__()
4
5          self.critic = nn.Sequential(
6              nn.Linear(num_inputs, hidden_size),
7              nn.ReLU(),
8              nn.Linear(hidden_size, 1)
9          )
10
11         self.actor = nn.Sequential(
12             nn.Linear(num_inputs, hidden_size),
13             nn.ReLU(),
14             nn.Linear(hidden_size, num_outputs),
15             nn.Softmax(dim=1),
16         )
17
18     def forward(self, x):
19         value = self.critic(x)
20         probs = self.actor(x)
21         dist  = Categorical(probs)
22         return dist, value
```

Figure 8. Creation of the deep neural network.

The cumulative return can be calculated at each step with a simple function that receives the reward for a given value of γ, which in this case has been set to 0.99 (Fig. 9). The values are accumulated by keeping track of previous results.

```
1 def compute_returns(next_value, rewards, masks, gamma=0.99):
2     R = next_value
3     returns = []
4     for step in reversed(range(len(rewards))):
5         R = rewards[step] + gamma * R * masks[step]
6         returns.insert(0, R)
7     return returns
```

Figure 9. Function for cumulative return calculation.

The next step is to configure the model characteristics. The number of inputs and outputs, hidden layers of the deep network and their learning rate, the type of model, i.e. Actor-Critic, and the optimizer to be used are set. In our case, we have used the Adan variant of the gradient descendent (Fig. 10).

```
1 def compute_returns(next_value, rewards, masks, gamma=0.99):
2     R = next_value
3     returns = []
4     for step in reversed(range(len(rewards))):
5         R = rewards[step] + gamma * R * masks[step]
6         returns.insert(0, R)
7     return returns
```

Figure 10. Adjustment of model features.

Finally, the training of the model is performed. This is done within a cycle that calls the functions and evaluates each parameter at each step (Fig. 11).

```
1  state = envs.reset()
2
3  while frame_idx < max_frames:
4
5      log_probs = []
6      values    = []
7      rewards   = []
8      masks     = []
9      entropy = 0
10
11     for _ in range(num_steps):
12         state = torch.FloatTensor(state).to(device)
13         dist, value = model(state)
14
15         action = dist.sample()
16         next_state, reward, done, _ = envs.step(action.cpu().numpy
                                                           ())
17
18         log_prob = dist.log_prob(action)
19         entropy += dist.entropy().mean()
20
21         log_probs.append(log_prob)
22         values.append(value)
23         rewards.append(torch.FloatTensor(reward).unsqueeze(1).to(
                                                        device))
24         masks.append(torch.FloatTensor(1 - done).unsqueeze(1).to(
                                                        device))
25
26         state = next_state
27         frame_idx += 1
28
29         if frame_idx
30             test_rewards.append(np.mean([test_env() for _ in range(
                                                       10)]))
31             plot(frame_idx, test_rewards)
32
33     next_state = torch.FloatTensor(next_state).to(device)
34     _, next_value = model(next_state)
35     returns = compute_returns(next_value, rewards, masks)
36
37     log_probs = torch.cat(log_probs)
38     returns   = torch.cat(returns).detach()
39     values    = torch.cat(values)
40
41     advantage = returns - values
42
43     actor_loss  = -(log_probs * advantage.detach()).mean()
44     critic_loss = advantage.pow(2).mean()
45
46     loss = actor_loss + 0.5 * critic_loss - 0.001 * entropy
47
48     optimizer.zero_grad()
49     loss.backward()
50     optimizer.step()
```

Figure 11. Model training cycle.

3. RESULT AND DISCUSSION

As indicated above, we used the CartPole-v0 by OpenAI playground as an evaluation platform for the algorithm. Many experiments were performed on this platform for different configurations of the algorithm, which guarantees the stability and scalability of the results. In all cases, the reward and its accumulated value throughout the training were recorded and plotted.

The code was developed on a machine with a single Intel Core i7-7700HQ eight-core 3.8 GHz CPU with 64-bit architecture and 24 GB of RAM. This machine runs a Debian Bullseye Linux OS with kernel 5.18.0. Within Google Colab we used Torch 1.11.0+cu113, gym 0.17.3, piglet 1.0.0, pyvirtualdisplay 3.0, python-opengl 3.1.0+dsfg-1, matplotlib 3.2.2, and numpy 1.21.6, configuring a GPU in the Runtime.

Fig. 12 shows four of our experiments with 16 environments, 256 hidden layers in the deep neural network, and a learning rate of 3e-4. The average run time of each experiment was 2 minutes and 16 seconds. The cumulative reward in the experiment in Fig. 12(a) was 186, in the experiment in Fig. 12(b) it was 200, in the experiment in Fig. 12(c) it was 112, and in the experiment in Fig. 12(d) it was 134.

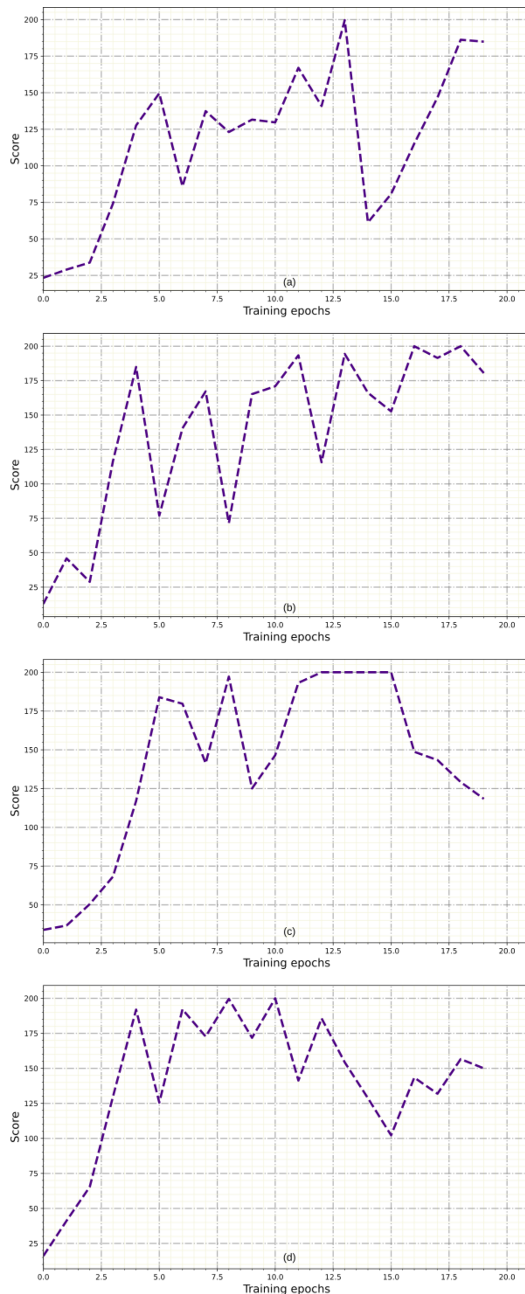

Figure 12. Comparison of learning in four experiments with 16 environments, 256 hidden layers in the deep neural network, and 20 epochs: (a) case with 186 cumulative rewards, (b) case with 200 cumulative rewards, (c) case with 112 cumulative rewards, and (d) case with 134 cumulative rewards.

The first two experiments in Fig. 12 perform better in reaching a higher cumulative reward and maintaining a high reward value throughout the training than the last two cases. In addition, the latter two cases tend to deteriorate their behavior after epoch 15. To evaluate the effect of the parameters on the algorithm, similar training was performed by varying only one of these parameters, first the number of training epochs, then the number of environments, and finally the depth of the neural network. In the first set of experiments, the number of training epochs was doubled, and the result is shown in Fig. 13. The cumulative reward in the experiment in Fig. 13(a) was 200, in the experiment in Fig. 13(b) it was 171, in the experiment in Fig. 13(c) it was 200, and in the experiment in Fig. 13(d) it was 200. The average time per experiment was four minutes and 34 seconds.

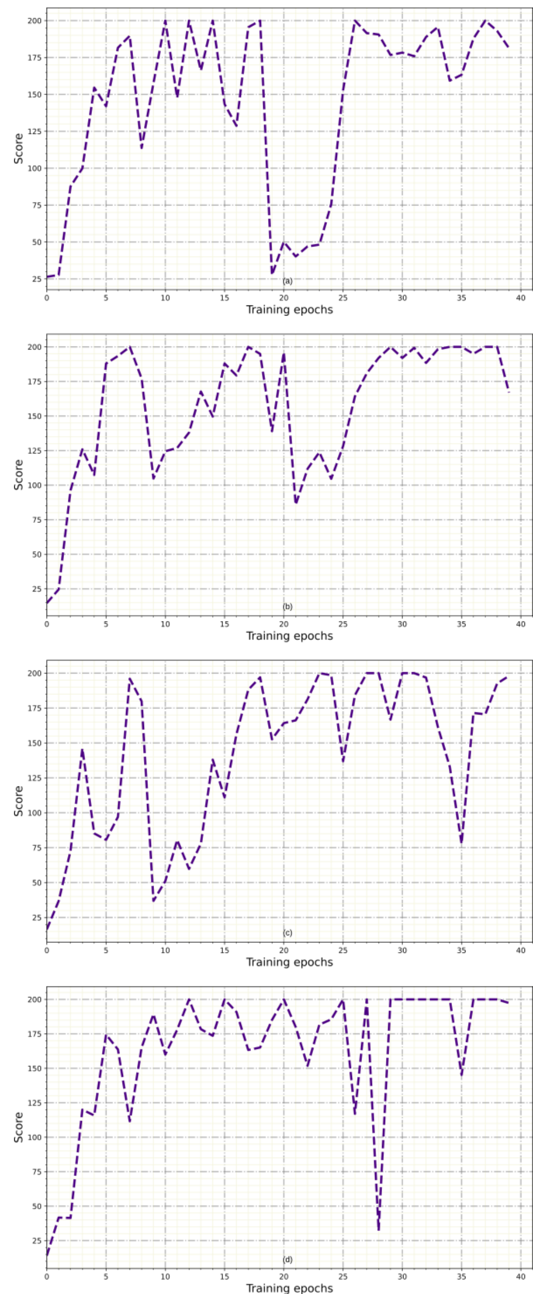

Figure 13. Comparison of learning in four experiments with 16 environments, 256 hidden layers in the deep neural network, and 40 epochs: (a) case with 200 cumulative rewards, (b) case with 171 cumulative rewards, (c) case with 200 cumulative rewards, and (d) case with 200 cumulative rewards.

By doubling the training time, a better behavior of the models is observed. In many of the experiments it is observed that at some point the reward decreases, but somehow this contributes to increasing the stability of the process, since the reward value increases again, and remains high for the rest of the training.

The third set of experiments again reduced the training time to 20 epochs, and doubled the number of environments to a total of 32, keeping all other parameters constant. The results are shown in Fig. 14, in the experiment in Fig. 14(a) the cumulative reward was 110, in the experiment in Fig. 14(b) the cumulative reward was 200, in the experiment in Fig. 14(c) the cumulative reward was 200, and in the experiment in Fig. 14(d) the cumulative reward was 200. The average time in these experiments was two minutes and 40 seconds.

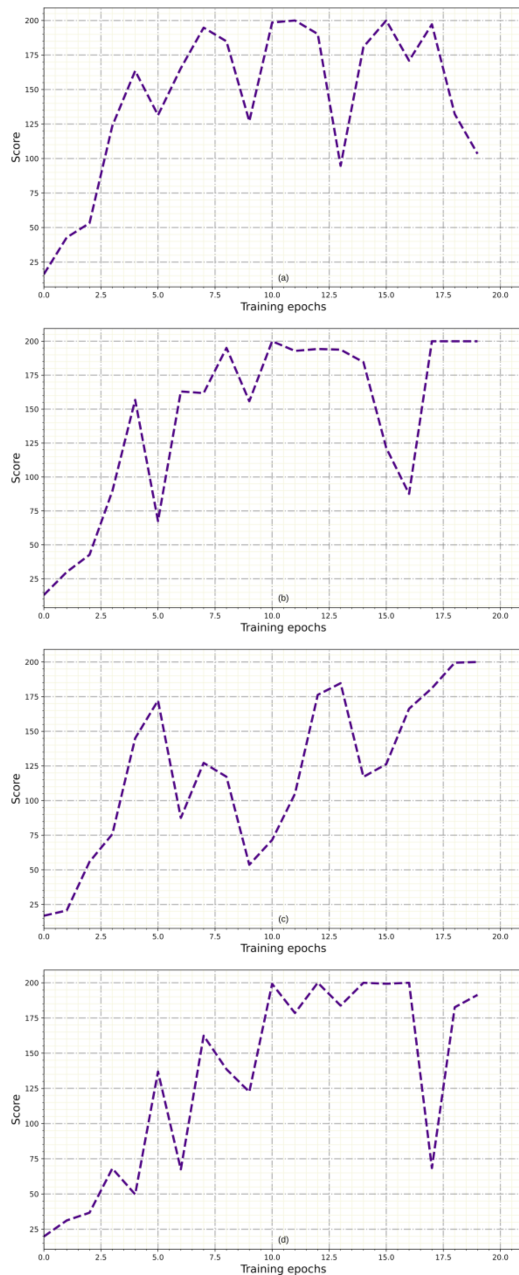

Figure 14. Comparison of learning in four experiments with 32 environments, 256 hidden layers in the deep neural network, and 20 epochs: (a) case with 110 cumulative rewards, (b) case with 200 cumulative rewards, (c) case with 200 cumulative rewards, and (d) case with 200 cumulative rewards.

In these experiments, it is observed that the increase in the number of environments has an impact on the stabilization of the reward, since unlike the experiments in Fig. 12, the reward value remains more stable, and the final cumulative reward values also increase. The time cost for half of the environments is only 17.6% additional.

The last set of experiments evaluates the impact of neural network depth, returning to the 16 environment configuration, and increasing the number of hidden layers to 512. All other model parameters are kept the same. Examples of these experiments are shown in Fig. 15, in the experiment in Fig. 15(a) the cumulative reward was 98, in the experiment in Fig. 15(b) the cumulative reward was 200, in the experiment in Fig. 15(c) the cumulative reward was 200, and in the experiment in Fig. 14(d) the cumulative reward was 86 The average time in these experiments was two minutes and six seconds.

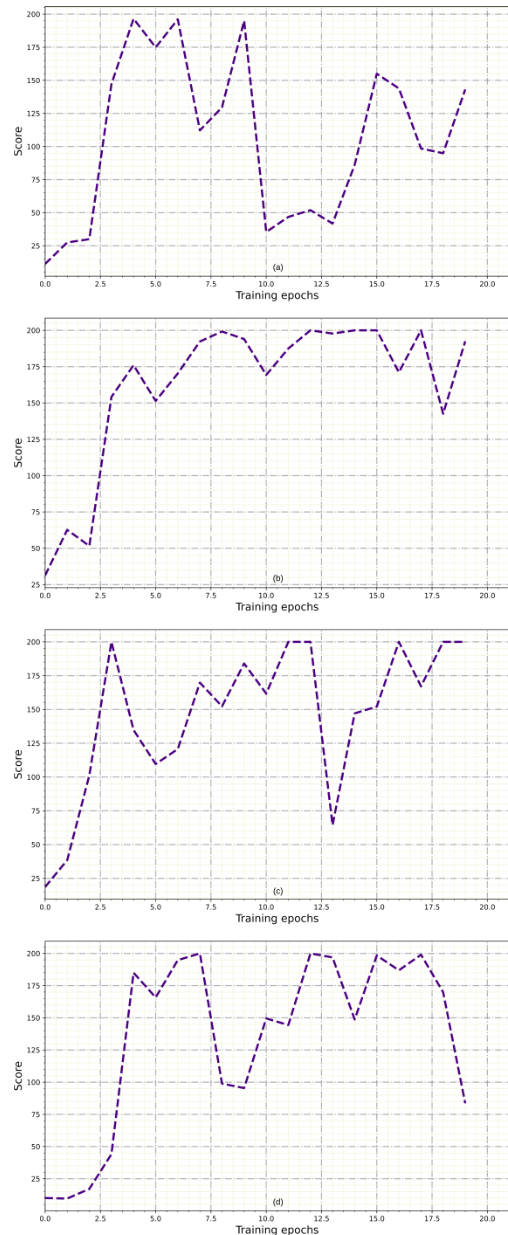

Figure 15. Comparison of learning in four experiments with 16 environments, 512 hidden layers in the deep neural network, and 20 epochs: (a) case with 98 cumulative rewards, (b) case with 200

cumulative rewards, (c) case with 200 cumulative rewards, and (d) case with 86 cumulative rewards.

The increase in the complexity of the deep neural network has an interesting effect on the performance of the agent, as it increases its performance. This is observed in a small reduction in the total training time (7.3% reduction), which indicates that it is easier for the agent to learn in its interaction with the environment. Even so, the overall performance in terms of cumulative reward is similar to that observed in the earlier experiments with a shallower neural network.

4. CONCLUSION

This paper implements and evaluates the Advantage Actor-Critic (A2C) algorithm to determine its actual performance for different combinations of parameters running on a single multi-core CPU system without GPU. The algorithm was implemented in Python on the Google Colab platform, making use of its GPU service, and with PyTorch support. The sensitive parameters of the algorithm were number of agents/environments, neural network depth, and training duration. From the results of multiple experiments it was concluded that the variables with the greatest impact on performance are those that improve the level of interaction of the agent with its environment. In particular, longer training times have a significant impact on the stability and final value of the accumulated reward. The depth in the neural network also facilitates the learning of the environment, and the number of environments helps to stabilize the reward behavior along the process. Therefore, it can be stated that a higher level of interaction of the agent in the environment significantly increases its level of learning. In addition, since the algorithm is faster and more robust than classical RL algorithms, it is more suitable than other similar techniques for use in embedded systems, even more so if one considers that it can be used in both discrete and continuous space problems.

5. DECLARATIONS

Authors declare that they have no conflict of interest in this research paper.

6. ACKNOWLEDGMETS

This work was supported by the Universidad Distrital Francisco José de Caldas, in part through CIDC, and partly by the Facultad Tecnológica. The views expressed in this paper are not necessarily endorsed by Universidad Distrital. The authors thank the research group ARMOS for the evaluation carried out on prototypes of ideas and strategies.

7. REFERENCES

[1] J. Watson, “Psychology as the behaviorist views it,” Psychological Review, vol. 20, no. 2, pp. 158–177, 1913.

[2] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, “A general reinforcement learning algorithm that masters chess, shogi, and go through self-play,” Science, vol. 362, no. 6419, pp. 1140–1144, dec 2018.

[3] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, “Grandmaster level in StarCraft II using multi-agent reinforcement learning,” Nature, vol. 575, no. 7782, pp. 350–354, oct 2019.

[4] B. Kiumarsi, K. G. Vamvoudakis, H. Modares, and F. L. Lewis, “Optimal and autonomous control using reinforcement learning: A survey,” IEEE Transactions on Neural Networks and Learning Systems, vol. 29, no. 6, pp. 2042–2062, 2018.

[5] J. Han, A. Jentzen, and W. E, “Solving high-dimensional partial differential equations using deep learning,” Proceedings of the National Academy of Sciences, vol. 115, no. 34, pp. 8505–8510, 2018.

[6] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, “Learning agile and dynamic motor skills for legged robots,” Science Robotics, vol. 4, no. 26, p. 5872, 2019.

[7] P. A. Erdman and F. Noé, “Identifying optimal cycles in quantum thermal machines with reinforcement-learning,” npj Quantum Information, vol. 8, no. 1, p. 1, 2022.

[8] C. M. Wu, E. Schulz, T. J. Pleskac, and M. Speekenbrink, “Time pressure changes how people explore and respond to uncertainty,” Scientific Reports, vol. 12, no. 1, p. 4122, 2022.

[9] S. Manna, T. D. Loeffler, R. Batra, S. Banik, H. Chan, B. Varughese, K. Sasikumar, M. Sternberg, T. Peterka, M. J. Cherukara, S. K. Gray, B. G. Sumpter, and S. K. R. S. Sankaranarayanan, “Learning in continuous action space for developing high dimensional potential energy models,” Nature Communications, vol. 13, no. 1, p. 368, 2022.

[10] J. Aznar-Poveda, A.-J. García-Sánchez, E. Egea-López, and J. García-Haro, “Approximate reinforcement learning to control beaconing congestion in distributed networks,” Scientific Reports, vol. 12, no. 1, p. 142, 2022.

[11] E. Kuprikov, A. Kokhanovskiy, K. Serebrennikov, and S. Turitsyn, “Deep reinforcement learning for self-tuning laser source of dissipative solitons,” Scientific Reports, vol. 12, no. 1, p. 7185, 2022.

[12] Z. cheng Qiu, G. hao Chen, and X. min Zhang, “Trajectory planning and vibration control of translation flexible hinged plate based on optimization and reinforcement learning algorithm,” Mechanical Systems and Signal Processing, vol. 179, no. 109362, p. 109362, 2022.

[13] R. T. Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith, “Reward machines: Exploiting reward function structure in reinforcement learning,” Journal of Artificial Intelligence Research, vol. 73, no. 1, pp. 173–208, 2022.

[14] L. Bobadilla, F. Martinez, E. Gobst, K. Gossman, and S. M. LaValle, “Controlling wild mobile robots using virtual gates and discrete transitions,” in 2012 American Control Conference (ACC). IEEE, 2012, pp. 743–749.

[15] F. Martínez, F. Martínez, and E. Jacinto, “Performance evaluation of the nasnet convolutional networkin the automatic identification of covid-19,” International Journal on Advanced Science Engineering Information Technology, vol. 10, no. 2, pp. 662–667, 2020.

[16] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," ICML 2016, 2016.

[17] B. Sellami, A. Hakiri, S. B. Yahia, and P. Berthou, "Energy-aware task scheduling and offloading using deep reinforcement learning in SDN-enabled IoT network," Computer Networks, vol. 210, no. 1, p. 108957, 2022.

[18] A. Biswas, P. G. Anselma, and A. Emadi, "Real-time optimal energy management of multimode hybrid electric powertrain with online trainable asynchronous advantage actor–critic algorithm," IEEE Transactions on Transportation Electrification, vol. 8, no. 2, pp. 2676–2694, 2022.

[19] P. Zhao, X. Li, S. Gao, and X. Wei, "Cooperative task assignment in spatial crowdsourcing via multi-agent deep reinforcement learning," Journal of Systems Architecture, vol. 128, no. 1, p. 102551, 2022.

[20] H. Yue, J. Liu, D. Tian, and Q. Zhang, "A novel anti-risk method for portfolio trading using deep reinforcement learning," Electronics, vol. 11, no. 9, p. 1506, 2022.

[21] H. Montiel, F. Martínez, and F. Martínez, "Parallel control model for navigation tasks on service robots," Journal of Physics: Conference Series, vol. 2135, no. 1, p. 012002, 2021.

[22] K. Kanno and A. Uchida, "Photonic reinforcement learning based on optoelectronic reservoir computing," Scientific Reports, vol. 12, no. 1, p. 3720, 2022.