The Impact of Task Complexity on Agent Training Efficiency in Reinforcement Learning Scenarios

Rufat Mammadzada
Faculty of Information Technologies and Control, Department of Computer Engineering
Azerbaijan State Oil and Industry University
Baku, Azerbaijan
mammadzadarufat@gmail.com

Rahim Mammadzada
Faculty of Information Technologies and Control, Department of Instrumentation Engineering
Azerbaijan State Oil and Industry University
Baku, Azerbaijan
rahim.mammadzada@gmail.com

***Abstract*—This study investigates the effects of increasing task complexity on the training and performance of intelligent agents in various reinforcement learning scenarios. Beginning with a basic navigation task where the agent has precise knowledge of object coordinates, progressively challenging scenarios that test the agent's adaptability and decision-making abilities are introduced. Heuristics and sensory mechanisms, such as ray-casting and real-time feedback, are utilized along with the division of the task into smaller episodes to evaluate how agents optimize their performance under different conditions. The scenarios range from enhanced navigation with additional movement axes to sensor-based navigation without direct coordinate knowledge, and finally to dynamic environments with multiple rewards, time constraints, and competitive interactions between agents acting as hunters and prey. Each scenario is designed to incrementally increase complexity, requiring the agent to employ more sophisticated strategies and adapt Its behavior in response to new challenges. The findings provide insights into the scalability of reinforcement learning techniques and the ability of agents to handle increasingly complex tasks, offering valuable implications for agent learning in complex scenarios.**

Keywords—Reinforcement Learning, Unity3D, Scenario, Agent, Reward

I. INTRODUCTION

Task decomposition in reinforcement learning (RL) is a valuable strategy to improve learning efficiency and performance. By breaking down complex tasks into simpler subtasks, agents can focus on relevant portions of the state space for each sub-goal, leading to faster learning and better outcomes [1].

This division allows for parallel learning of individual subgoals, minimizing the overall learning time. Automatic partitioning of state/input spaces can exploit the distinct characteristics of different regions and agents, enabling efficient learning without prior knowledge of the domain [2].

In multiagent scenarios, such as territory division in hide-and-seek games, hierarchical learning structures have demonstrated the ability to optimize task performance by tailoring strategies for specific roles, such as seeking and hiding [3]. Segmenting tasks into smaller parts significantly improves training efficiency and success rates.

In 1950s "Samuels Checkers" program implemented the first alpha-beta pruning algorithm. Similarly to reinforcement learning algorithms, he implemented a loss function that calculates the probability of winning the game based on the current position. The main ideas that gave rise to Reinforcement learning arose in the 1980s. The first game to implement a reinforcement earning algorithm is considered TD backgammon using the TD (Lambda) algorithm [4] and [5].

Since then, computers and graphics cards have become powerful, paving the way for the development of more functional game engines like Unity3D.

The Unity ML-Agents Toolkit was introduced in 2017 as an open-source project, allowing developers and researchers to apply reinforcement learning and other machine-learning techniques within the Unity environment [6] and [7].

Two Algorithms that can be chosen from in Unity Ml-Agents PPO and SAC, along with the physics abilities of the Unity game engine and the visualization aspect, make it an ideal environment to simulate scripts where agents can learn to navigate around different dynamic and static environments and increase performance [8]. The learning task can be divided into smaller steps, thus allowing the agent to develop a better learning pattern.

II. METHODOLOGY

Unity3D is used for game development. However, over time more researchers have used it as a simulation environment. This is due to its simplicity, built-in libraries, and access to 3rd party packages. Some researchers integrate game logic into their papers to make the research more entertaining. In this work, a simple scenario will be chosen and gradually complexity is increased. With the mix of researcher and developer logic. Eventually, a rival agent is added to see how it affects the training results [9].

In multi-agent scenarios, such as territory division in hide-and-seek games, hierarchical learning structures have demonstrated an ability to optimize task performance by tailoring strategies for specific roles, like seeking and hiding [3].

By segmenting tasks into manageable components, training efficiency and success rates can be significantly improved. Since then, computer and graphics cards have become stronger paving the way for developing more functional game engines like Unity3D.

The Unity ML-Agents Toolkit was officially introduced in 2017 as an open-source project, allowing developers and researchers to apply reinforcement learning and other machine-learning techniques within the Unity environment [6]. Two Algorithms that can be chosen in Unity Ml-Agents PPO and SAC, along with the physics abilities of Unity game engine and the visualization aspect make it an ideal environment to simulate scripts where agents can learn

to navigate around different dynamic and static environments and increase the performance [10].

III. Generated and Trained Scenarios

A. Scenario 1: Avoid and Gather

This scenario explores how an agent trains in a basic virtual environment. In the scene, several game objects represent the player, the goal, and the obstacles. Collusions are detected using Unity's Rigidbody component. However, to detect each object, they must be differentiated. Thus, Unity's tag system is used: the sphere representing the goal is assigned a "Reward" tag, while obstacles, such as the walls, are assigned a "Punishment" tag. This setup is shown in Figure 1 below.

Figure 1: Agent and the reward.

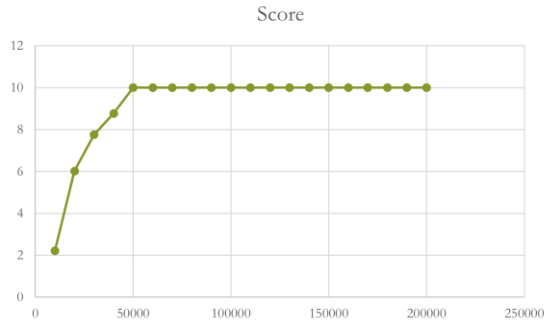
When setting up the agent itself, the behavior parameters of the agent need to be defined first. To do this, it is necessary to logically determine which input parameters the agent can learn from. In this case, if the agent knows both the coordinates of the goal and the object, it can understand where it is headed. Since in Unity3D, each coordinate is represented as a Vector3, which is a structure containing 3 floats, it can be concluded that there will be 6 input parameters for its behavior. Only 1 continuous value will be sufficient for the output.

After logically establishing the size of the network, another script is written to inherit from the ML-Agents package's agent class. Here, functions for setting up the agent for training can be overridden. The three main functions are "OnEpisodeBegin,","CollectObservations,"and "OnActionReceived".

With "OnEpisodeBegin," the scene resets by specifying the circumstances under which the episode ends. This is set by the user, who controls when the agent is penalized or rewarded. "CollectObservations" is used to assign the input parameters, while "OnActionReceived" is applied using the neural network's output.

The previously mentioned tag system can be utilized for ending the episode and handling rewards. This can be achieved using the "OnTriggerEnter" function, which detects any contact between objects. Within this function, specific tags of the objects can be checked by introducing if statements. If an object with a "Punishment" tag is hit, a negative reward is received. Similarly, a positive reward is obtained if an object with a "Reward" tag is hit. After setting up these conditions, the agent was trained in this environment, and the results are shown in Table.

From Graph 1 it is seen, that the agent initially did not know how to obtain the reward. However, after learning that reaching the reward object's coordinates results in a reward, it began to recognize the winning pattern and learned to consistently move towards the reward. The main challenge for the agent here was understanding the concept of boundaries. Since the agent did not have a direct way to perceive the walls, it had to comprehend that the reward was located within an enclosed space.

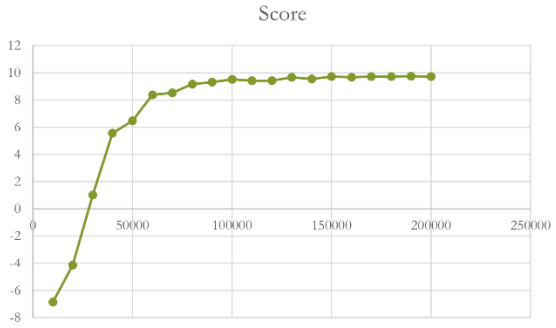


Graph 1: Scenario1 Score-Epoch diagram.

B. Scenario 2: Avoid and Gather 2

In this scenario, the intelligent agent continually holds a precise knowledge of the coordinates for both the wall and the sphere. However, the agent has less restrained movement capabilities: it moves to the left or right along the X-axis, as well as forward and backward along the Z-axis. Introducing an additional axis does not impact the setup that much, these are only differences compared to previous scenario.

Firstly, the neural network output size increases to 2 inside the behavior parameters. Secondly, inside the agent subclass script, a second output value is needed to move the object in both dimensions.



Graph 2: Scenario2 Score-Epoch diagram.

This enhanced movement flexibility requires the agent to develop a more sophisticated navigation strategy. It must balance its actions carefully to reach the sphere while avoiding collisions with the walls. The agent's ability to rotate around the Z-axis introduces an additional layer of complexity, allowing for more nuanced adjustments in navigation and positioning. This is represented in Graph2, where a similar graph of the 1st scenario is depicted, albeit with more fluctuations.

C. Scenario 3: Avoid and Gather With Sensor

In this scenario, there is no longer direct access to the coordinates of objects. Instead, the agent relies on a sensory mechanism that casts rays in front of it to detect nearby items.

With the loss of direct coordinate information, the agent must strategically use its ray-perception sensor to gather real-time sensory data and interpret the type and position of objects based on the tags detected by the rays.

This sensory feedback allows the agent to adjust its movements and rotations dynamically. The challenge is to devise an effective navigation strategy that balances exploration and careful maneuvering to achieve the objective.

Figure 2: Agent at the environment with sensory element

These rays are equipped with the ability to identify objects based on specific tags, such as "Sphere" or "Walls," depending on what is within their path. This is done with the help of the ray perception sensor component which comes with the Unity ML-agent package. It is shown at Figure 2.

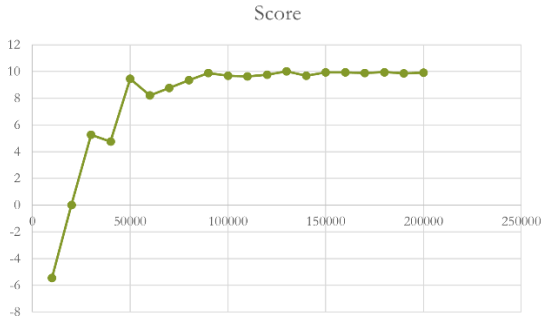
Ray Perception Sensor in Unity ML-Agents is a tool that simulates an agent's ability to detect objects in its environment by casting rays (like LIDAR). It's often used for tasks like obstacle avoidance and navigation. This component has some key parameters such as:

- Rays Per Direction: Number of rays cast in each direction.
- Ray Length: The maximum distance each ray can reach.
- Detectable Tags: Tags for objects the rays can detect.
- Field of View: The spread angle of the rays, configurable for a full 360° or a narrower range.

This element gives an advantage as it allows the agent to detect multiple tagged objects. It also makes it possible to move agents into a completely unknown environment, where even a wall in a random position might be. One thing to note here is when the "Ray Perception Sensor" is used, there is no need to change the size of input parameters in behavior parameters, nor add any parameters with code inside the overwritten "Collect Observations" functions. This component does all that automatically, and in the Unity's inspector, the ray directions can be adjusted, numbers, color, the maximum angle of ray projection, and the sphere's cast radius, which defines the contact points for ray detection [11].

Additionally, the agent's movement has also changed, to benefit the use of rays. Now the agent can move forward or backward, instead of moving left or right, and it can rotate around its Z-axis to adjust its orientation. Despite the change in perception and movement, the end goal remains the same.

Graph 3 shows us the results of training with the new setup. As in the first two scenarios, it took some time before understanding the assignment.



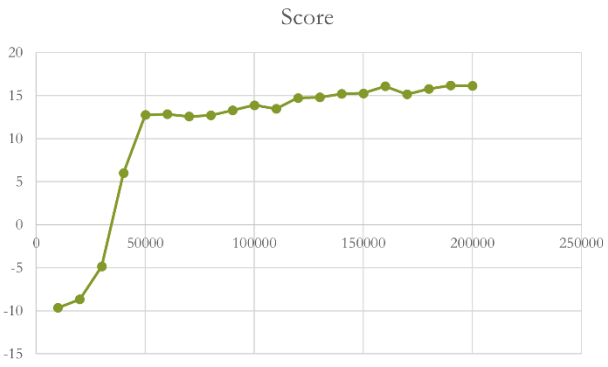
Graph 3: Scenario 3 Score-Epoch diagram.

Some fluctuations and spikes are seen in the results. This is due to the increased number of parameters that need to be processed by the agent. For each ray cast by this component, the neural network gets input parameters for its distance and if there is a specified tag of the object too.

D. Scenario 4: Avoid and Gather With Multiple Rewards

In this scenario, multiple rewards were added. To do this, a "prefab" was created with the tag "award" and instantiated several times. Additionally, visual feedback was made as a ground color change to red or green for one second Figure 3.

Figure 3: Agent at the environment with multiple rewards



Graph 4: Scenario 4 Score-Epoch diagram

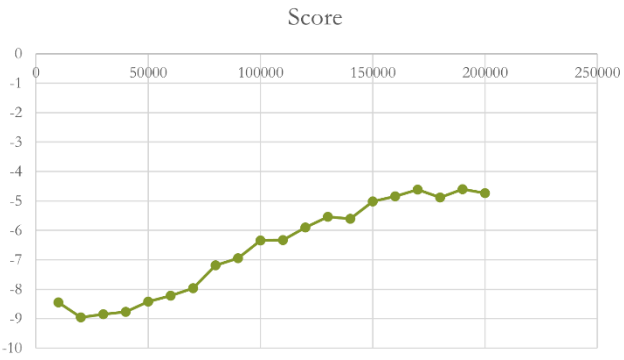
Here the use of the "Ray Perception Sensor" helps us to keep several input parameters of the neuron network unchanged. If we still used the coordinates of each reward, then it would complicate the agent's learning process more for each added reward.

Like the previous scenarios, the agent started in the negative region, gradually learned to evade walls, and collect rewards. To learn agent needs to collect all the rewards and avoid the wall. After some time, it reaches near the maximum possible reward with a slight fluctuation at Graph 4.

E. Scenario 5: Avoid and Gather With Time Limit

In this scenario, a timer that punishes the agent after a certain period passes. The goal here is to force the agent to speed up the reward-gathering process resulting in a better score. This timer is called and restarted inside the overwritten "On Episode Began" function. Additionally, the rewards spawn function was altered with a nested for loop for reattempting spawning if an object is too close to the player.

During the training process, the agent's parameters were specifically set so that, it would be impossible to collect all the rewards in the allotted period. This was done to see if the agent would continue to improve even in a rigged scenario. The results of this training are shown in Graph 5 below.



Graph 5: Scenario 5 Score-Epoch diagram

From this graph, it is seen that the agent always remained at negative value, due to the task being impossible to complete. However, its score continued to improve. At first, it learned how to dodge obstacles and pick the nearest target.

Afterwards, it also reduced the delay before rotating when collecting a reward since this improved the next reward detection time.

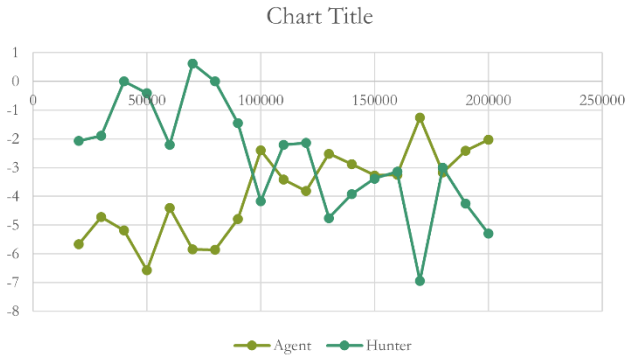
It had a slight impact on the score improvement.

E Scenario 6: Catch and Run

In this scenario, another agent as the hunter was added, and now 1st agent is in the role of the prey. Both agents must evade the walls and get punished when the timer runs out. However, when it comes to reward, the prey must collect it in a limited time while the agent hunter tries to catch it. If a hunter catches the prey, it loses. The same is true when the prey gathers rewards while avoiding the hunter. Additionally, when one agent succeeds, the other gets a penalty Figure 4.

Here, hunters' movement and perception are identical to the prey. There are a couple of differences. Firstly, the hunter does not need to interact with the reward or, it would prioritize destroying rewards over hunting the prey. To make the scenario fair and get achieve better results, a condition is set for the hunter to spawn at a certain distance from the player.

Figure 3: Two Agent at the environment where the back is the hunter, and the purple one is the prey



Graph 6: Scenario 6 Score-Epoch diagram

Graph 6 shows that the prey agent had more trouble at first because it had to simultaneously avoid the walls, the hunter agent, and collect rewards, while the hunter agent only had to dodge the walls and focus on the prey. After some training, both started evading the walls, and the prey learned to stay away from the hunter and focus on the rewards. Once the prey agent had the upper hand, the hunter agent started to make different strategies.

Among these strategies were: waiting for prey near the reward, following the prey's trajectory as it headed toward the reward, and several others. At this point, the prey would begin paying even more attention to the hunter, leading to better results at the end of the training session.

IV. Conclusion

In summary, the simplicity of an environment directly impacts the stability of an agent's performance and the rewards it can achieve. When the environment is simpler, with fewer variables and more predictable outcomes, the agent's learning and optimization processes become simpler and more stable. This leads to more consistent performance and higher reward attainment.

Conversely, complex environments introduce increased variability and uncertainty, disrupting the learning process and making reward acquisition less stable. The agent must navigate through more intricate scenarios, which can lead to fluctuating performance and potentially lower overall rewards.

This principle is illustrated through various scenarios:

Scenario 1: The agent operates along a single axis, where performance remains consistently high with minimal fluctuations. This reflects the stability afforded by a simpler environment.

Scenario 2 and Scenario 3: These scenarios show slight fluctuations in performance but still exhibit relatively stable outcomes, demonstrating that even with some complexity, stability is maintained to a certain degree.

Scenario 4: Here, the agent experiences fluctuating performance despite learning from episodes.

This variability leads to less consistent reward acquisition, highlighting the challenges of more complex environments with multiple rewards.

Scenario 5: In this scenario, with the surplus of the rewards for the given time constraint. The agent's learning process is more gradual, which results in improved performance over time. This suggests that persistence, stability, and higher rewards can be achieved even in more complex settings.

Scenario 6: Initially, the hunter agent successfully catches the prey agent. However, the dynamics

change when the prey agent learns to avoid the hunter agent.

The prey agent begins to receive more rewards, reflecting the evolving nature of complex interactions and the constant adjustments required by the agent.

Findings suggest that when the scenarios become more complex agents can learn albeit with more time. In other scenarios involving complex structures like robots or systems with joints, each joint motion can be trained separately, and the results can be connected for faster training while reducing the complexity.

REFERENCES

[1] Karlsson, J. (1994). Task Decomposition in Reinforcement Learning. https://api.semanticscholar.org/ CorpusID:10262719.

[2] Sun, R. & Peterson, T. (1999). Multi-agent Reinforcement Learning: Weighting and Partitioning. Neural Networks: The Official Journal of the International Neural Network Society,

12(4-5)727–753. doi:10.1016/s0893-6080(99)00024-6.

[3] Gunady, M.K., Gomaa, W. & Takeuchi, I. (2012). Multi-agent Task Division Learning in Hide-and-Seek Games. In Artificial Intelligence: Methodology, Systems, Applications.

[4] Pignede, T. (2012). Evolution of Reinforcement Learning in Games or How to Win against Humanswith Intelligent Agents. PhD thesis. http://www.ias.informatik.tu-darmstadt.de/uploads/Teaching/

AutonomousLearningSystems/Pignede_ALS_2012.pdf.

[5] Konen, W. (2015). Reinforcement Learning for Board Games: The Temporal Dierence Algorithm. CorpusID:64222607.

[6] Juliani, A., Berges, V.-P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M. & Lange, D. (2020). Unity: A General Platform for Intelligent Agents. arXiv preprint arXiv:1809.02627v2. San Francisco, CA 94103 USA. https://arxiv.org/abs/1809.02627.

[7] Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. (2017). Proximal Policy Optimization Algorithms

[8] Litwynenko, K. & Plechawska-Wójcik, M. (2021). Analysis of the possibilities for using machine learning algorithms in the Unity environment. Journal of Computer Sciences Institute, 20, 197–204–. doi:10.35784/jcsi.2680

[9] Wang, T., Gupta, T., Mahajan, A., Peng, B., Whiteson, S. & Zhang, C. (2020). RODE: Learning Roles to Decompose Multi-Agent Tasks. ArXiv, abs/2010.01523. https://api.semanticscholar.org/CorpusID:222132819.

[10] Xu, C., Zhu, R. & Yang, D. (2021). Karting racing: A revisit to PPO and SAC algorithm. 2021 International Conference on Computer Information Science and Artificial Intelligence (CISAI), 310–316. https://api.semanticscholar.org/CorpusID:247179117.

[11] Kozlov, D.A. & Myasnikov, V. (2022). The Impact of a Set of Environmental Observations in theProblem of Acquiring Movement Skills in Three-Dimensional Space Using Reinforcement Learning Algorithms. In Proceedings of the 2022 VIII International Conference on Information Technology and Nanotechnology (ITNT) (pp. 1–5). IEEE. doi:10.1109/ITNT55410.2022.9848598.