Project 2.1 - AI and Machine Learning

Controlling Agents in the Unity Game Engine

Francisco Costa (i6349732), Ioannis Panayiotou (i6357037), João Tavares (i6366229), Luuk Janssens (i6332099), Michał Urbanowicz (i6354317), Tom Daugherty (i6358492), Vilmos Udvari (i6354710)

January 20, 2024

Contents

Abstract

Reinforcement learning (RL), a transformative branch of artificial intelligence, has driven advancements in robotics, finance, and logistics. However, its application in virtual gaming environments offers a unique opportunity to explore the development and behavior of agents in highly dynamic and complex scenarios. This report delves into the design, training, and optimization of RL agents within Unity-based game simulations, investigating how environment complexity and training parameters affect performance. By analyzing these factors, this study highlights best practices for creating adaptive agents. These insights have potential applications beyond gaming in real-world problem-solving domains.

1 Introduction

Reinforcement learning (RL) in gaming environments serves as a valuable platform for advancing artificial intelligence research. Games provide dynamic and complex scenarios that mimic real-world challenges while offering the advantage of visual feedback to facilitate the interpretation of results. The lessons learned from developing RL agents in games are highly transferable to other fields, such as robotics, logistics, and decision-making systems.

This report focuses on the development and optimization of an RL agent trained to play SoccerTwos, a virtual soccer game included in the Unity ML-Agents Toolkit[1]. This open-source toolkit, developed by Unity, enables games and simulations to function as environments for training intelligent agents, making it a cornerstone for RL experimentation.

The SoccerTwos environment features a field with two goals, a ball, and four players divided into purple and blue teams, each with two agents. The goal is to design an RL agent capable of mastering the game's dynamics and successfully scoring against opponents.

To address compatibility issues, the version of the ML-Agents Toolkit used in this project is a forked repository that resolves key problems in the original implementation. Detailed information about this fork and its modifications can be found in the project's repository[2].

2 Methods

To approach the problem realistically, several modifications were made to the SoccerTwos environment to better simulate plausible gameplay scenarios and analyze agent performance.

2.1 Sensor Adjustments

In the original SoccerTwos environment, each player was equipped with two sets of raycast sensors—one pointing forward and the other backward. These sensors, which emit rays in a conical shape, gather information about surrounding objects and their distances, serving as the "eyes" of the agents. To enhance realism, the backward sensor was removed, as it would be analogous to having vision at the back of the head.

2.2 Head Movement Sensor

A further modification was introduced to allow players to control their head movement independently of their body orientation. This enables players to look in directions other than where their bodies are facing, providing the agent with greater flexibility in decision-making. This environment serves as a testbed to evaluate the impact of enhanced sensory input and control mechanisms on agent performance.

2.3 Hyperparameter Experiments

To investigate the influence of hyperparameters on training, a series of controlled experiments was conducted. Agents were trained multiple times while varying a single parameter per experiment, such as learning rate or neural network size. During these experiments, system resource usage (CPU, GPU, and memory) was monitored using a custom Python script. The performance of the resulting models was measured using the self-play ELO metric which was

[1] `https://github.com/Unity-Technologies/ml-agents`

[2] `https://github.com/luuk7/mlai`

automatically recorded by the ml-agents toolkit during training. The results of these experiments were analyzed to assess the effects of each hyperparameter on training efficiency and agent performance.

3 Implementation

3.1 Sensors

To implement the head-turning sensor, a new class, 'AgentSoccerHeadTurn', was created, inheriting from the default 'AgentSoccer' class. This design allows for targeted modifications while preserving the core functionalities of the original agent.

The key changes include:

- Adding new attributes to define the parameters and conditions for head rotation.
- Implementing a 'MoveHeadInput' function to manage head rotation.
- Extending the 'MoveAgent' function to incorporate head movement alongside body control.

The primary logic for head movement is encapsulated in the 'MoveHeadInput' function. The pseudocode for this function is shown below:

```
Input: Head rotation axis

If headRotateAxis equals 0:
    return

Determine rotationDirection:
    If headRotateAxis equals 1:
        rotationDirection = clockwise
    Else If headRotateAxis equals 2:
        rotationDirection = counterClockwise

Rotate head in rotationDirection

If head exceeds max rotation angle:
    Revert rotation
```

This mechanism ensures controlled head rotation within predefined limits, enhancing the agent's ability to gather directional input in a realistic manner.

3.2 Monitoring

A Python script was developed to monitor resource usage during agent training. The script utilizes the `psutil` and `pynvml` libraries to measure CPU, memory, and GPU usage. It automates the logging of relevant processes, which is crucial when training involves multiple environments that spawn numerous processes.

The script captures process details by name and stores the data in a structured log file for later analysis. This approach enables consistent monitoring of resource consumption, providing insights into the computational demands of training sessions.

CPU usage is measured as the sum of the utilization percentages of all available cores. This means the value ranges from 0 to $100 \times$ (number of CPU cores). The GPU usage measurement provided by the `pynvml` library differs from other implementations, such as that of the Windows Task Manager. `pynvml` reports the value as a combination of the usage of all components. Consequently, the utilization of a GPU operating at 100% of its CUDA cores for training might be reported as a significantly lower value.

4 Experiments

Multiple training runs were conducted to examine the impact of different hyperparameters on the performance of the resulting models and the efficiency of the training process. A total of six distinct hyperparameters were tested across 17 training runs, each consisting of at least 15 million steps, necessitating over 90 hours of training. In some cases, the resource usage data was only recorded for the first 15 million steps, but the run itself was longer.

4.1 Simulation Environment

All training runs were performed using an executable compiled from the same scene[3], which was based on

[3] `https://github.com/luuk7/mlai/blob/phase-3/Project/Assets/ML-Agents/Examples/SoccerNew/Scenes/HeadTurn.unity`

the default SoccerTwos scene provided in the ML-Agents Toolkit. The only modification involved replacing the backwards ray sensors with our head-turning implementation.

4.2 Tested Hyperparameters

Hyperparameter	Tested values
Batch size/buffer size	2048/20480, 4096/40960, 8192/81920
Gamma	0.95, 0.99, 0.995
Number of hidden units	256, 512, 1024
Learning rate	0.0001, 0.0003, 0.001
Number of environments	4, 10, 16
Play against latest model ratio	0.5, 0.8

Table 1: Tested hyperparameters and their respective values.

Table 1 lists the six hyperparameters examined and their respective tested values. Batch size and buffer size were tested together, with the buffer size consistently set to ten times the batch size. This approach was adopted to mitigate potential issues arising from insufficient buffer capacity. For analysis purposes, this pairing was treated as a single hyperparameter.

4.3 Testing Setups

Due to time constraints, data collection had to be parallelized. Therefore, each hyperparameter was tested on a different system.

Table 2 shows the relevant information about the different systems the hyperparameters were tested on. Due to availability and driver constraints, the CPU was used for training in the tests for the following hyperparameters:

- Number of hidden units
- Learning rate
- Play against latest model ratio

The GPU was used instead for the following:

- Gamma
- Batch size/buffer size
- Number of environments

5 Results

5.1 Resource Usage

The timeseries graphs for the CPU and GPU usage metrics were extremely noisy since the values varied rapidly during testing. Therefore, the averages from the runs are shown instead for the sake of clarity and ease of analysis.

5.1.1 Number of hidden units

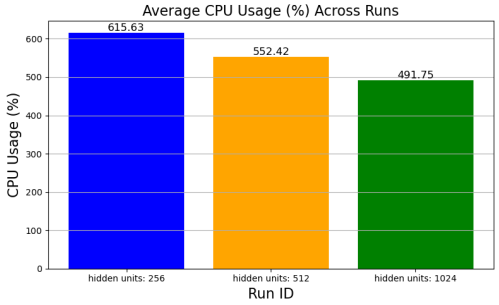

Figure 1: Average CPU usage (%) for different hidden unit configurations across runs.

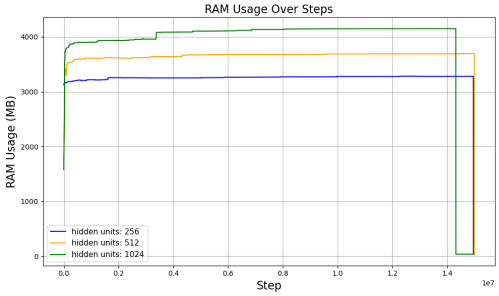

Figure 2: RAM usage (MB) for different hidden unit configurations across runs.

5.1.2 Learning rate

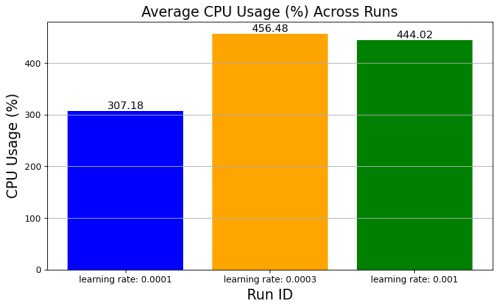

Figure 3: Average CPU usage (%) across training runs for the learning rate hyperparameter.

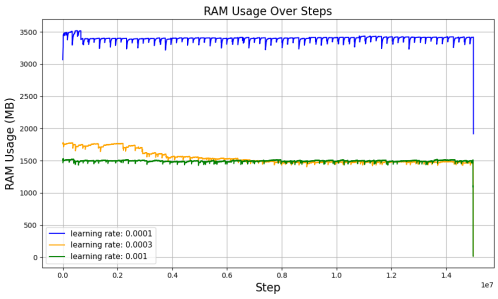

Figure 4: RAM usage (MB) across training runs for the learning rate hyperparameter.

5.1.3 Play-against-latest-model ratio

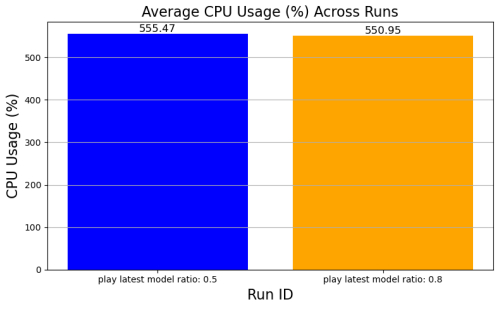

Figure 5: Average CPU usage (%) for the play-against-latest-model ratio hyperparameter across training runs.

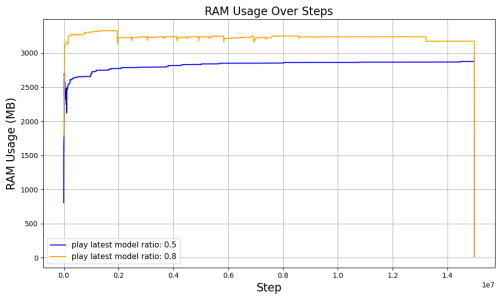

Figure 6: RAM usage (MB) for the play-against-latest-model ratio hyperparameter across training runs.

5.1.4 Gamma

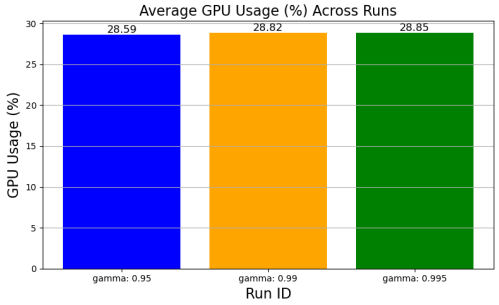

Figure 7: Average GPU usage (%) across training steps for the gamma hyperparameter.

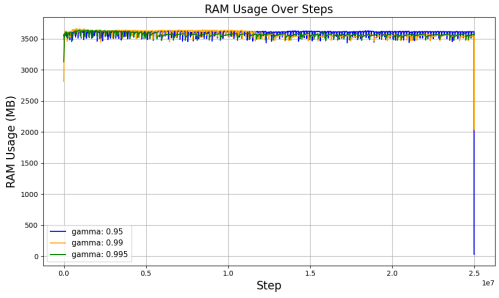

Figure 8: RAM usage (MB) across training steps for the gamma hyperparameter.

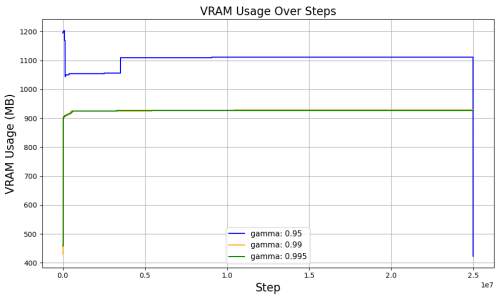

Figure 9: VRAM usage (MB) across training steps for the gamma hyperparameter.

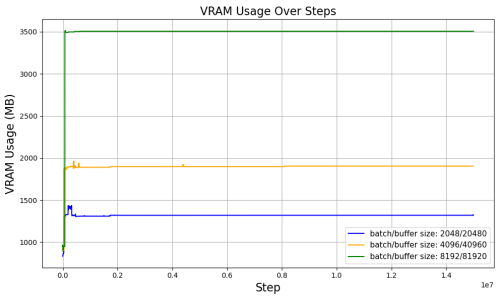

Figure 12: VRAM usage (MB) across training steps for the batch size and buffer size hyperparameters.

5.1.5 Batch size and buffer size

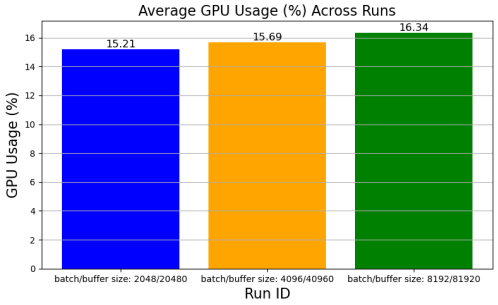

Figure 10: Average GPU usage (%) across training steps for the batch size and buffer size hyperparameters.

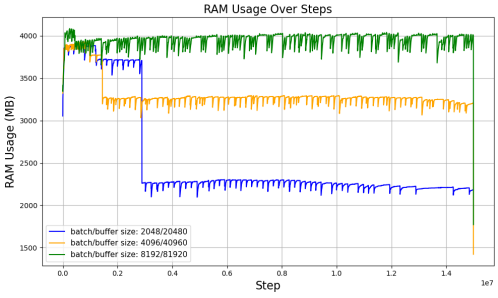

Figure 11: RAM usage (MB) across training steps for the batch size and buffer size hyperparameters.

5.1.6 Number of environments

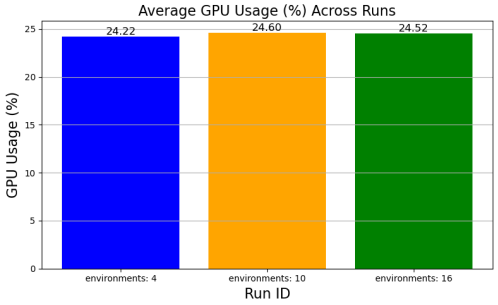

Figure 13: Average GPU usage (%) across training steps for the number of environments hyperparameter.

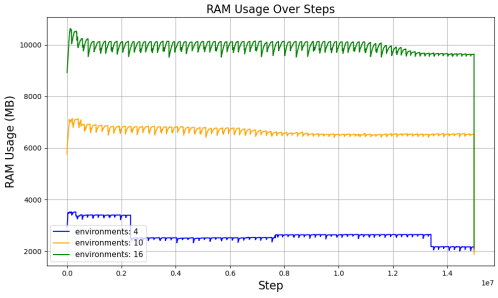

Figure 14: RAM usage (MB) across training steps for the number of environments hyperparameter.

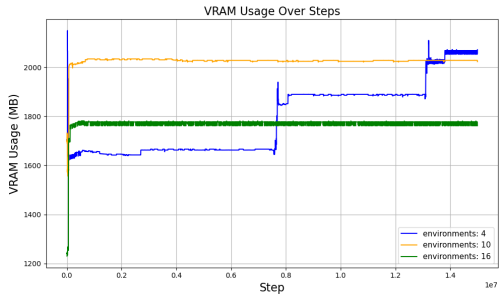

Figure 15: VRAM usage (MB) across training steps for the number of environments hyperparameter.

Figures 1-14 were generated using the `matplotlib` Python library from the data obtained during the experiments.

5.2 Performance

Figure 16: Progression of ELO across training steps for the gamma hyperparameter.

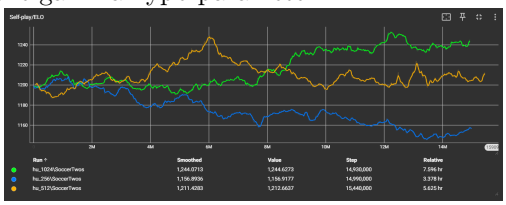

Figure 17: Progression of ELO across training steps for the number of hidden units hyperparameter.

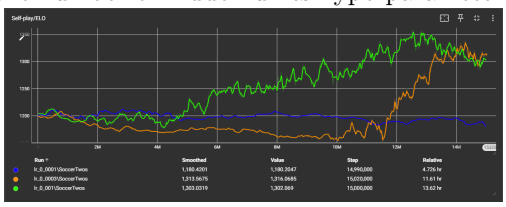

Figure 18: Progression of ELO across training steps for the learning rate hyperparameter.

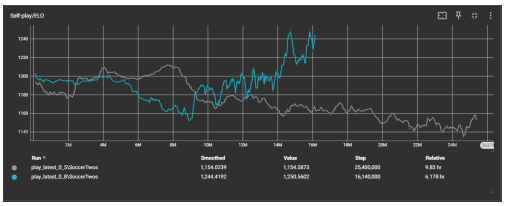

Figure 19: Progression of ELO across training steps for the play-against-latest-model ratio hyperparameter.

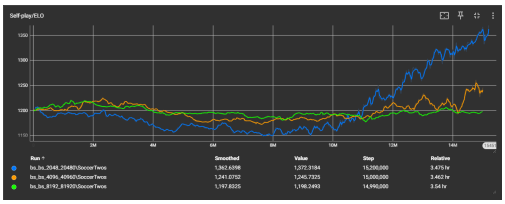

Figure 20: Progression of ELO across training steps for the batch size and buffer size hyperparameters.

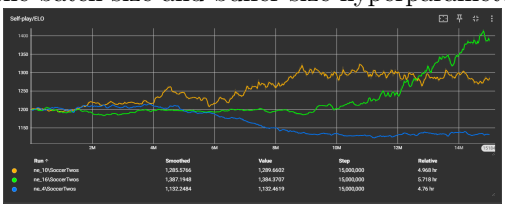

Figure 21: Progression of ELO across training steps for the number of environments hyperparameter.

Figures 16-21 were generated using the tensorboard implementation provided by the ML-Agents Toolkit.

6 Discussion

6.1 Results

6.1.1 Number of hidden units

Increasing the number of hidden units increases performance, as demonstrated by improved ELO scores in Figure17. This comes at the cost of an increase in RAM usage, but interestingly a decrease of approximately 25% in CPU usage compared to 256 hidden units. The decrease in CPU usage could be caused by the increase in training duration when increasing the number of hidden units.

6.1.2 Learning rate

A low learning rate (e.g. 0.0001) increased the performance with higher stability and overall ELO, as plotted in Figure18. This comes at the cost of an average increase of 133% in RAM usage, but led to approximately 31% lower CPU usage, compared to the other learning rates. Again, there was also an increase in training duration as the learning rate was lowered.

6.1.3 Play-against-latest-model ratio

A higher play-against-latest-model ratio (0.8) demonstrates overall better performance, as seen in Figure19. Resource wise not much difference is shown. The play-against-latest-model ratio of 0.8 shows a decrease of less than 1% in CPU usage and an increase in RAM usage of approximately 16%.

6.1.4 Gamma

Using the gamma value of 0.995 results in the best overall performance, both in highest ELO and fastest growing ELO. A gamma value of 0.95 shows the ELO is under the starting value even after 30 million steps, shown in Figure16. Gamma seems to have a big impact on performance. Resource-wise, the gamma hyperparameter does not seem to have a big impact on GPU usage, as can be seen in Figure 7. The VRAM only differs for the gamma value of 0.95, increasing approximately 5%.

6.1.5 Batch Size and Buffer Size

Larger batch sizes result in a smoother ELO curve, decreasing the ELO maximum. The best performing value for batch/buffer size is 2048/20480, showing a significantly higher ELO at 15 million steps in Figure20. Increasing the batch/buffer size increases the GPU usage slightly, but significantly increases the RAM and VRAM usage by approximately 77% and 169% respectively.

6.1.6 Number of environments

Increasing the number of environments (from 4 to 16) significantly increase the final ELO value at 15 million steps with approximately 22%. However, it also increases the training duration by almost 1 hour (Figure21). In terms of resource usage, the GPU usage does not change significantly, but the RAM shows a big increase of more than 300%. The VRAM graph in Figure 15 shows inconsistent values, which raises uncertainty about the data.

6.2 Challenges and findings

6.2.1 Parallelized testing

Executing the tests in different environments introduces differences in resource metrics and overall performance. The lack of GPU support for some tests could have severe impact on the results.

6.2.2 Findings

The results suggest the importance of balancing computational efficiency with configurations. Configurations like moderate batch sizes, lower learning rates, deeper networks, higher play-against-latest-model ratio, precise gamma and the right number of environments can help in efficient RL training.

6.3 Future improvements

6.3.1 Parallelized testing

In future experiments, it is suggested to use the same environment and system specifications for more concise results.

6.3.2 Broader hyper-parameter tuning

To optimize the testing configuration even further and explore new configurations, a larger range of tested hyper-parameters should be used. Also, hyperparameters should be tested simultaneously, to see if the combinations yield different results.

7 Conclusion

We analyzed reinforcement learning (RL) algorithms using the Unity ML-Agents framework in the SoccerTwos environment. The experiments tested various parameter settings, such as batch size, buffer size, learning rate, and network architecture, revealing their significant impacts on training efficiency and the use of computational resources. Additionally, the inclusion of a head-turning sensor notably enhanced agents' spatial awareness and adaptability, improving decision-making capabilities. The experiments were conducted using compiled executables from a modified SoccerTwos scene in the ML-Agents Toolkit.

Overall, we found that while larger networks and higher learning rates can accelerate learning, they also come with increased computational demands. These findings emphasize the importance of balancing computational efficiency with performance optimization when designing RL systems.

However, the findings should be interpreted with caution. The study did not examine the interaction of hyper-parameters due to time constraints, which means that simultaneous changes to multiple parameters could lead to unexpected outcomes. Furthermore, the non-deterministic nature of the physics engine and variations caused by different hardware may result in irreproducible results, introducing an element of chance. As a result, repeating an experiment could yield different outcomes.

References

A Hardware Specifications

Hyperparameter	**Device**	**CPU**	**RAM (GB)**	**GPU**	**VRAM (GB)**
Batch size/buffer size	Lenovo LOQ 15APH8	AMD Ryzen 5 7640HS @ 4.30 GHz	16	NVIDIA GeForce RTX 4050	6
Gamma	Lenovo MT 83DH	AMD Ryzen 7 8845HS @ 3.80 GHz	32	NVIDIA GeForce RTX 4070	0.44
Number of hidden units	MacBook Pro	M3 Pro 12-Core	36	N/A	N/A
Learning rate	Dell G5 5590	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.21 GHz	NVIDIA GeForce RTX 2060	7.9	N/A
Number of environments	Custom PC	AMD Ryzen 7 3700X	16	NVIDIA GeForce RTX 2080	8
Play against latest model ratio	MacBook Pro 13in 2022	M2 10-Core	16	N/A	N/A

Table 2: Specifications of the systems the hyperparameters were tested on.