Effects of curriculum learning on maze exploring DRL agent using Unity ML-Agents

Master of Science Thesis
University of Turku
Department of Computing
2022
Aki Tervo

UNIVERSITY OF TURKU
Department of Computing

Aki Tervo: Effects of curriculum learning on maze exploring DRL agent using Unity ML-Agents

Master of Science Thesis, 64 p.
June 2022

As the amount of studies on the usage of machine learning in video games has increased, few of these studies use curriculum learning. This thesis aims to show the benefits that curriculum learning, even in an unoptimized state, can provide to deep reinforcement learning when used with Unity ML-Agents toolkit. This thesis contains two case studies of machine learning agents going through a maze. Both of the case studies have two `Agent`s: one which uses curriculum learning and one which does not. First case study has the `Agent`s use their inbuilt `Vector Sensor` and in the second case study they use `Raycast Perception Sensor`. The data that is gathered from the case studies is from the training of two `Agent` types and the evaluation of the `Agent`s. The results show that adding curriculum learning can increase the stability of training and improve the results of the evaluation. On the other hand, the training and evaluation results are unstable which makes getting definitive results impossible.

Keywords: Unity, curriculum learning, deep reinforcement learning, ML-Agents toolkit

TURUN YLIOPISTO
Tietotekniikan laitos

Aki Tervo: Effects of curriculum learning on maze exploring DRL agent using Unity ML-Agents

Pro gradu -tutkielma, 64 s.
Kesäkuu 2022

Videopeleissä hyödynnettävää koneoppimista käsittelevien tutkimusten määrä on jatkanut kasvamista, mutta yhtä koneoppimisen osa-aluetta käytetään näissä tutkimuksissa harvoin: opetussuunnitelman mukaista oppimista. Tämän tutkielman tavoitteena on osoittaa opetussuunnitelman käytön hyötyä syvävahvistusoppimiseen Unity ML-Agents-työkalupakissa, vaikka kyseinen opetussuunnitelma ei ole optimoitu. Tässä tutkielmassa on kaksi tapaustutkimusta, joissa on kaksi koneoppimisagenttia. Näiden agenttien tehtävä on löytää maalialue sokkelosta. Toisella agentilla on opetussuunnitelma käytössä. Ensimmäisessä tapaustutkimuksessa agentit käyttävät ML-Agents-työkalupakin agenteille sisäänrakennettua sensoria nimeltään `Vector Sensor` ja toisessa tapaustutkimuksessa agentit käyttävät sensoria nimeltään `Raycast Perception Sensor`. Tapaustutkimuksissa data kerätään agenttien koulutuksesta ja evaluaatiosta. Kerätyt tulokset osoittavat, että opetussuunnitelman mukaisen oppimisen lisääminen voi parantaa agenttien koulutuksen vakautta ja evaluaatiossa saavutettuja tuloksia. Toisaalta molemmissa tapaustutkimuksissa agenttien koulutus on epävakaata, mikä tekee opetussuunnitelman mukaisen oppimisen hyötyjen tarkan määrittelyn mahdottomaksi.

Asiasanat: Unity, opetussuunnitelman mukainen oppiminen, syvävahvistusoppiminen, ML-Agents toolkit

Contents

1 Introduction

The purpose of this thesis is to explore the effect of adding unoptimized curriculum learning to a machine learning agent that has unoptimized hyperparameters. The Unity project with which the study of this thesis is made can be found in GitHub [1].

As a person who intends to enter the game development industry, I have had interest in video game AI. One of discontent for video game players is the poor quality of the game's AI. One way to improve the video game AI that has been offered is the usage of machine learning for the creation of the AI. This method promises a way to create more complex and realistic behaviors. [2]

While this promise served as the initial spark of inspiration for this thesis, the work that I have done for the creation of my own game served as the source from which the concept of this thesis was born. As an independent solo developer there are no hard deadlines for the publishing of my game, I can take as long as I want to optimize the machine learning agent, but this is not true in more professional video game development settings.

Often in professional video game development experience what is called "crunch", which means a drastic overtime that goes for an extended period of time. This practice can lead developers to make suboptimal choices in development. If these developers use machine learning for a video game, this "crunch" can lead them into trying to take shortcuts which can be poor optimization of the machine learning

algorithm. This is why it is important to study how well machine learning algorithms function in different settings when they are not optimized to those settings.

This thesis is structured into seven chapters and their descriptions are as follows:

- Chapter 2 has topics that are important to know in order to gain a better understanding of the topic of this thesis.
- Chapter 3 describes how the training of the agents is made and how the training and evaluation environments function.
- Chapter 4 consists of two case studies that have data from the training and the evaluation. The contents of these case studies differ only in the way that the machine learning agent in question gathers data from its surrounding environment.
- Chapter 5 has the analysis for the results of the training and evaluation and it is also divided into two case studies like Chapter 4.
- Chapter 6 has my thoughts and opinions that came up during the making of this thesis along with topics that could be pursued for further studies.
- Chapter 7 contains the final thoughts and a wrap up of this thesis.

2 Background

In this chapter, I go through certain topics that can help to provide background information that are important to understand the topic of this thesis. These topics are the game development environment used: Unity, machine learning, deep learning, reinforcement learning, deep reinforcement learning, curriculum learning and the Unity ML-Agents toolkit.

2.1 Unity

Unity, previously also known as Unity3D, is a popular game engine that has been used by both professional and amateur games. One of the reasons for Unity's popularity is its license. The license allows anyone to use Unity until their revenue and/or funding reaches 100,000 US dollars. Only after that must the user upgrade from the personal license. [3] This along with a large gallery of assets found on Unity's Asset Store and the large amount of tutorials and other help materials has also helped to increase Unity's popularity.

Even though Unity is mainly known for its use as a game engine, it has also found usage in other fields. These fields range from industry to films and animation all the way to architecture. Alongside these fields Unity is also used as a research environment, most known for research on augmented reality, virtual reality and machine learning. Some of this research is done by Unity Labs.

2.2 Machine learning

According to M. I. Jordan and T. M. Mitchell [4] machine learning is a discipline that focuses on two questions: "How can one construct computer systems that automatically improve through experience? and What are the fundamental statistical computational-information-theoretic laws that govern all learning systems, including computers, humans, and organizations?". Machine learning is nowadays used in speech recognition, natural language processing, controlling robots, data processing and other applications.

The large amount of different use cases for machine learning has been a driving force in the creation of the different forms of machine learning. Many of the machine learning algorithms used are focused on function approximation problems and the machine learning algorithm's task is to improve itself in a way that improves the function's accuracy.

There are three major paradigms of machine learning. These are supervised, unsupervised and reinforcement learning. Out of these three paradigms, the interest of this thesis is on reinforcement learning as that is the one paradigm that is used the most when machine learning is studied with video games.

In order to learn, the machine learning algorithms can be given a volume of training data. This data is in the form of paired input-output training samples with the data labeled. When the data is in this form, the paradigm used is supervised learning. [5] When the data is not labeled then the machine learning paradigm is unsupervised learning. Reinforcement learning will be discussed in more detail later in this chapter.

2.3 Deep Learning

Deep learning is a type of representation learning. Representation learning refers to methods that do not need their training data to be in the input-output pairs. The training data can be given in a raw form from which the machine learning algorithm can automatically start to classify these input-output pairs. When these representation learning methods are stacked to multiple layers then deep learning is born. An important and intriguing aspect of deep learning is that these representation learning layers are not designed by humans. The layers emerge from the data through the learning procedure.

It is due to this, that deep learning algorithms can solve problems that have been problematic to solve with previous machine learning algorithms and deep learning algorithms have managed to improve the results in areas like image and speech recognition along with many other fields of study and usage. A more detailed description of deep learning can be found in a paper by LeCun et al. called *Deep learning* [6].

One other description for deep learning comes from IBM [7]. According to IBM: "Deep learning is a subset of machine learning, which is essentially a neural network with three or more layers." The purpose of these neural networks is to emulate the human brain thus allowing the algorithm to, in a sense, learn from the data it is given. Based on these descriptions of deep learning, it is easy to conclude that classical machine learning algorithms that are created with deep learning belong to the unsupervised paradigm but deep learning has also been combined with semi-supervised learning.

2.4 Reinforcement Learning

According to a book *Reinforcement Learning: An Introduction* [8] reinforcement learning is a paradigm of machine learning alongside supervised and unsupervised

learning. It can be summarized as a computational approach of learning from interactions. With it the machine learning has no knowledge of the actions it has to do and must discover the best series of actions to maximize its rewards via trial-and-error.

One of the bigger challenges of using reinforcement learning is the trade-off between the agent trying out new sequences of actions and the agent utilizing a sequence of actions that it has already learned. If the agent does not try out new sequences it cannot find new and better solutions and on the other hand if it always tries something new a stable result is impossible. This dilemma between exploitation of existing action sequences and exploration of new sequences is one of the major aspects that differentiates reinforcement learning from the other two major machine learning paradigms.

The reinforcement learning agents have all these things: explicit goals, ability to gain data from their environment and can choose which actions to do. Alongside the agent and the environment there are four other main subelements of the reinforcement learning system. These are a policy, a reward signal, a value function and a model of the environment.

Policy's purpose is to define the agents behavior. In a sense it maps the environmental inputs to actions. A reward signal is in essence the goal in the reinforcement learning problem. In each step that happens in training the agent is sent a single number by the environment. That number is the reward that the agent tries to maximise. For this reason the agent should not have the ability to affect the generation of the reward signal in the environment.

The value function in reinforcement learning specifies the long term reward accumulation. While the rewards are an immediate good that the agent gains, value is the amount of reward that the agent can expect to gain in the future starting from a specific state. This allows the agent to choose an action that has a lower reward

so that it can take another action with much higher reward. Due to the nature of values they are much harder to define than rewards and that is why the most important component of reinforcement learning is an efficient method for estimating values.

The final subelement of reinforcement learning is the model of the environment, the purpose of which is to mimic the behavior of the environment. This model is used as a way to decide a course of action before it is experienced. This subelement is technically optional as there are both model-free and model-based methods of reinforcement learning.

Reinforcement learning is well suited for usage in machine learning studies regarding video games as video games serve as the environment for the agent and there always is a set of actions that can be given to the agent. Examples of reinforcement learning's usage with video games are the Arcade Learning Environment [9] and Unity ML-Agents toolkit [10]. Both of the systems can utilize reinforcement learning along with multiple different auxiliary learning methods like imitation learning.

2.5 Deep Reinforcement Learning

When the concept of deep learning is combined with reinforcement learning, deep reinforcement learning is created. According to Arulkumaran et al. in their paper *A brief survey of deep reinforcement* learning [11], deep reinforcement learning solves two major problems that are present with regular reinforcement learning algorithms: the lack of scalability and the limitation to low-dimensional problems due to memory, computational and sample complexity.

Deep reinforcement learning has opened the study of machine learning to research using machine learning in video games although this is often done in order to simulate the real world in order to see how well the machine learning agent can adapt to the real world. One major success for deep reinforcement learning happened when an

algorithm was created that could play a variety of video games from the game console Atari 2600. [12] The deep reinforcement learning agent could play most of the games better than any human could and also played almost always better than other reinforcement learning algorithms.

There are many different deep reinforcement learning algorithms but as this thesis uses a Proximal Policy Optimization algorithm that comes with Unity's ML-Agents toolkit as the machine learning algorithm of choice. I will focus on the Proximal Policy Optimization algorithm in this chapter but there are also other major deep reinforcement learning algorithms such as Soft Actor Critic and deep Q-learning.

Proximal Policy Optimization was created from policy gradient methods by OpenAI [13], [14]. According to OpenAI [13], these policy gradient methods, while providing good results in a variety of different environments, have major disadvantages. They often took a long time to learn simple tasks or they require a large amount of processing power.

Proximal Policy Optimization is called either as an algorithm or as a family of algorithms. No matter which definition is used, the key feature is the existence of a surrogate objective. The purpose of this surrogate objective is to regularize large policy updates in order to have each policy update step stay close to the previous-iteration policy. [15] To learn more about the Proximal Policy Optimization algorithm's inner workings, please see the paper by Schulman et al. called *Proximal Policy Optimization Algorithms* [14].

One of the most successful and high-profile uses for Proximal Policy Optimization has been when an agent created with it managed to defeat world champions from the video game Dota 2 in a best-of-three match. It took ten months of training for the agent to complete its training. [16]

2.6 Curriculum Learning

Curriculum learning arose with the advent of deep neural networks as deep neural networks were inspired from the human brain [17]. It also brought forth the question: What if we train machine learning models like we train a human? One of the first, if not the first to formalize easy-to-hard training strategies into what would be known as curriculum learning [17] was Bengio et al. [18]. The original paper on curriculum learning was not made with reinforcement learning in mind as it refers to training on sets of data [18].

The goal of curriculum learning is to make the training faster and more stable by usually starting the training from easier and smaller subset of the problem and then gradually expanding the subset until it is the whole training set. [18] There also exist strategies where this easy-to-hard strategy is reversed and is started from the harder and larger sets [19]. One of these methods is called hard example mining where the hardest and most difficult examples are used to train the model [20]. While these strategies might not be considered classic curriculum learning there have been studies where they have been categorized into the same family of strategies citeCLsurvey2.

One way to create a curriculum for reinforcement learning is proposed by Narvekar et al. in a paper called *Source Task Creation for Curriculum Learning* [21]. In the paper the method they propose, the curriculum is divided into source tasks. They then propose methods with which these source tasks can be found. These methods can provide a framework from which one can start designing a curriculum for a reinforcement learning agent.

Curriculum learning can be divided into two main categories: predefined and automatic curriculum learning. Automatic curriculum learning can then be divided into multiple subcategories. These subcategories are Self-Paced Learning, Transfer Teacher and Reinforcement Learning Teacher. In Self-Paced Learning the difficulty of training is changed depending on the failures of the trained model. In Transfer

Teacher there is an already trained model that selects the difficulty of the training for the trained model. In Reinforcement Learning Teacher, the "teacher" machine learning model adapts itself and the training based on the results of the trained model. There are also other methods of automatic curriculum learning that do not fall into these four categories. [19]

2.7 ML-Agents toolkit

Unity ML-Agents toolkit [10] is an open source project for Unity with which it is possible to train machine learning agents within the Unity game engine and create learning environments for those agents. The toolkit comes with two deep reinforcement learning algorithms which are Proximal Policy Optimization [22] and Soft Actor-Critic [23]. There are also two imitation learning algorithms: Behavioral Cloning [24] and Generative Adversarial Imitation Learning [25]. These can be used with the deep reinforcement learning algorithms. The toolkit also supports Self-Play [26] as well as curriculum learning [18].

The ML-Agents toolkit comes with a Python package that is made out of two components. First component is a low-level API. It can be used to directly interact with the Unity environment. The second component is an entry point that allows the training of the `Agents` in the Unity environment. [27] In terms of this paper the second component is more important.

The ML-Agents toolkit has three main components within the learning environment inside Unity:

1. *Academy.* The `Academy`'s purpose in the learning environment is to manage different parts of the toolkit within the environment. It keeps track of the steps taken in the simulation and manages the agents within it. [10] The `Academy` also serves as an interface between the learning environment inside

Unity and the Python API. It can store and change environmental parameters according to instructions written in a YAML-file that is used to configure the machine learning algorithm used. These environmental parameters can then be used to change aspects of the environment. For example environmental parameters could be used to make the gravity change every fixed interval. These environmental parameters are used to create the curriculum that can then be used in curriculum learning.

2. *Agent.* The `Agent` is the component that makes a `GameObject` in Unity into a machine learning agent. The `Agent` contains a labeled policy. This label is called a behavior name. If that label is not found in the specified YAML-file default settings for the training are used. This policy can be shared with any number of agents by having them share the behavior name. This makes the agents share the experience data during training. There can also be any number of different behavior names for different policies. This allows creation of scenes with multiple agents with different purposes. [10]

3. *Sensors.* Sensors are components that allow the `Agent` to gather data from the environment. Each `Agent` has an inbuilt way to gather observation data as a float through a `Vector Sensor`. Along with the `Vector Sensor` other sensors can be added for the `Agent`'s use. These sensors include `Raycast Perception Sensor` for both 2D and 3D, a `GridSensor`, a `CameraSensor` along with some other. The toolkit also allows for the creation of custom sensors for situations which are not covered by the sensors which come with the toolkit.

When using the ML-Agents toolkit a configuration for hyperparameters is required. Hyperparameters are parameters that are used to control the learning process. This configuration file is in YAML-format. In the configuration file the hyperparameters are divided into different categories, one of these categories is called

hyperparameters. In this thesis the term hyperparameters refers to this category of the configuration file. Some of these hyperparameters are universal in the ML-Agents toolkit while some others are tied to the deep reinforcement learning algorithm that the configuration file is for.

In this thesis the hyperparameters are the combination of the universal hyperparameters and the hyperparameters for the Proximal Policy Gradient algorithm. For the description of these hyperparameters see the documentation of the ML-Agents toolkit [28]. Their names are taken from the configuration file. The hyperparameters are:

1. Learning rate. Universal.
2. Batch size. Universal.
3. Buffer size. Universal.
4. Learning rate schedule. Universal.
5. Beta. Proximal Policy Optimization exclusive.
6. Epsilon. Proximal Policy Optimization exclusive.
7. Lambd. Proximal Policy Optimization exclusive.
8. Num epoch. Proximal Policy Optimization exclusive.

The first four of the hyperparameters are universal in ML-Agents toolkit and are present when either SAC or PPO algorithms are used. The beta, epsilon, lambd and num epoch hyperparameters are specific to the PPO algorithm.

By changing these hyperparameters one can affect the speed of learning or the randomness of the training. That is why it is important to find out good values for them in order for the `Agent` that one is creating to have the desired outcome

of training. For example, by increasing the value of the hyperparameter beta, the "randomness" of the `Agent`, called entropy, decreases more slowly.

In the Unity Editor there are three components that are required for the `Agent` to work. The descriptions of the components are based on the source code. These components are:

1. `BehaviorParameters`'s purpose is to set the `Agent`'s behavior and brain. With it one can set the amount of actions that the `Agent` can use and the amount of vector observations that the `Agent` can receive.

2. `DecisionRequester` automatically requests decisions for the `Agent` at regular intervals. It has two variables that can be changed in the Unity Editor. These are the Decision period which is the amount of Academy steps between decision requests and the other is a Boolean variable called Take Actions Between Decisions. If this variable is set to be true the `Agent` repeats its previous action until a new decision has been requested.

3. `Agent` is the actor that can observe its environment, and decide and execute actions based on the observations it has gathered. In order to create an `Agent`, one has to use it as a subclass for their own implementation of the `Agent`.

In order to follow how an active training session is going or how a completed training session went, TensorBoard can be used. It can be started from a Python terminal and then opened from a browser in order to view the statistics of the trained models. Unity ML-Agents toolkit saves the statistics of the trained model that can be viewed through TensorBoard, a TensorFlow utility. [29] With it, different statistics from the trained models can be viewed. These statistics are categorized into three main categories with two values that have their own categories. The main categories are:

1. Environment Statistics. These are statistics from the different environmental parameters that are used in the training. By default there are three values in this category: Cumulative Reward, which is the mean cumulative episode reward from all of the agents in training, Lesson, which shows when the model progresses to the next lesson during curriculum learning, and Episode Length, which is the mean amount of steps for each episode of all of the agents.

2. Policy Statistics. This category contains a maximum of eleven different values. The amount of values depends on which combination of machine learning algorithms are used in the trained model. For this thesis, the only important value is Entropy, which is the randomness of the model.

3. Learning Loss Functions. In this category there are a maximum of six values. The amount saved for viewing depends on the combination of machine learning algorithms.

3 Method

The goal for the study is to examine how well could curriculum learning improve different aspects of an Agent that is simple in terms of its code by learning to find its way through a maze with the least amount of steps. The aspects that curriculum learning could improve are the quality of its training and the results in an evaluation.

As I had some problems with the `Raycast Perception Sensor` provided with the ML-Agents toolkit during development I decided to create another version of the Agent, which is also tested in this thesis. To achieve this goal I created a learning environment with the help of a tutorial by Joseph Hocking [30]. This tutorial helped me to create an environment where a maze is generated whenever the Agent has finished a specified number of training episodes. For this study, the number of episodes was set to five.

The reason for choosing five as the number of episodes is that it allows the `Agent` to try each maze multiple times while still being a small enough number for the `Agent` to experience many different mazes during training. A goal is placed at the top right of the maze and the episode is completed, when the agent touches the goal or after 1,000 steps have been taken. The maximum number of steps was chosen as it is a large enough amount with which the `Agent` could clear the maze multiple times thus allowing it to experiment with each episode. The learning environment has eight instances of the Agent and each instance has a unique maze generated.

3.1 Agent

The environment has two versions of the `Agent`. The first one uses `Raycast Perception Sensor` as the only way to gather observations of the environment. This sensor is set to detect the walls of the maze and the goal object in straight lines that are aligned with the Z- and X-axis. This is demonstrated in Figure 3.1. It is also set to send five raycast results in a stack to the neural network allowing the `Agent` to have a sort of memory. The second version of the `Agent` uses a normalized 8-bit bitmask as a way to collect observations from its surroundings. The bitmask is created by checking if there is a wall or a goal next to the `Agent`. For each of them, a bit shifted value is then assigned and given to the sensor that comes with each `Agent` called `Vector Sensor`.

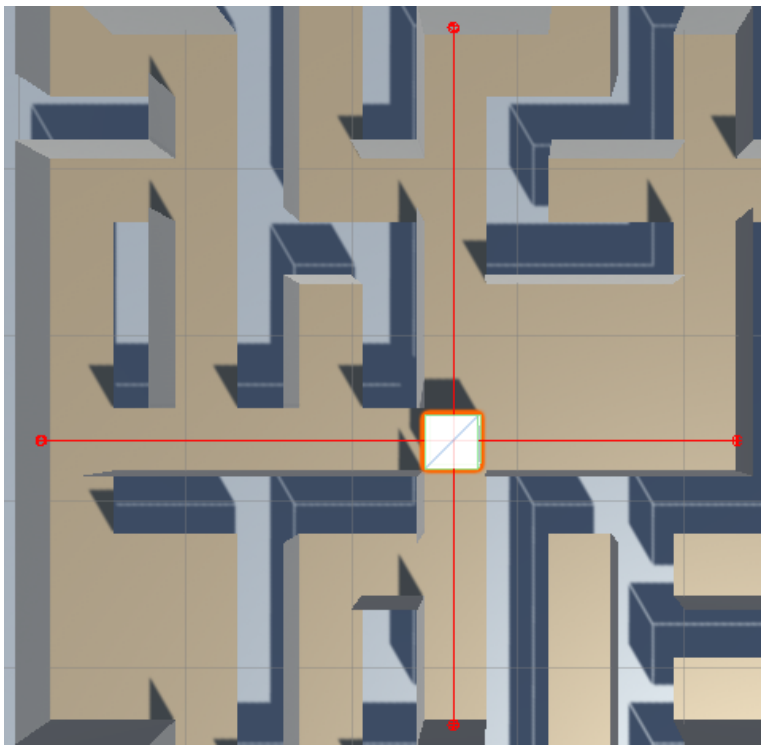

Figure 3.1: Illustration of the direction to which the `Raycast Perception Sensor` points its raycasts.

These results are then gathered and combined into a single variable, which is

then divided by 255 thus normalizing it. Inspiration for usage of the bitmask was taken from a paper by Goulart, Paes and Clua [31] where they tested different ways to collect the observations for an agent inspired by the game Bomberman. No other data is collected by the `Raycast Perception Sensor` using `Agent`. The bitmask using `Agent` also collects its current position in relation to the environment. The location data is not normalized. Although it could have improved the results, the changing size of the maze during curriculum learning makes normalization of the location data difficult and as such I decided that the improved results would not be worth it.

Both types of the `Agent` contain `BehaviorParameters` and `DecisionRequester` scripts. These two scripts are required for the successful usage of the Unity ML-Agents toolkit. `BehaviorParameters` component serves as the brain of the `Agent` it is connected to. The `BehaviorParameters`' purpose is to set the behavior of the `Agent` in the Unity Editor. In the `BehaviorParameters` user can set which trained model is used and set which of the three modes is used during runtime:

1. Default mode is for training a new model and if no training is detected resets to either of the two remaining modes.

2. Interference which only if `BehaviorParameters` has an active model in it as interference uses the given model to choose the actions of the `Agent`.

3. Heuristics which allows the user to control the `Agent` in a manner that has been specified in the `Agent's` code.

The `Agents` have only five possible actions to take: stay put, move forward, move back, move left and move right. These actions teleport the `Agent` one step to the specified direction. The length of the movement step is equal to the width of the maze's corridors. The reason for this type of movement is to keep the code reserved for movement as simple as possible and to use as few components for the `Agent`

as possible. This type of movement could make the learning of the `Agent` more unstable as the position of the `Agent` has more variance between each movement step.

The goal for the `Agents` is to reach a goal object in the maze. Once the goal is reached the `Agent` receives a reward which is calculated: `StepsUntilZero/MaxSteps`. `StepsUntilZero` is a variable that counts down from the maximum number of steps in an episode. Its value is reduced whenever a step is taken by the `Agent`. Reason for this formula is to tie the `Agent`'s reward to the speed at which it can clear the maze. This way the `Agent` can only receive bigger rewards by learning how to navigate the mazes more efficiently.

3.2 Learning Environment

The training scene in Unity contains one main `GameObject` called `Controller` that serves as the manager of all of the environments and has all of the learning environment instances as child objects. The `Controller` manages the creation of the mazes for each instance of the learning environment. A new maze is generated after the `Agent` has finished five episodes with its current maze. The maximum size of the mazes during learning process is 21 tiles in width and 19 tiles in height. I choose this size because this size allows complex enough mazes to provide challenge for the `Agent` while still allowing all of the different variations of the `Agent` to complete some of their training sessions successfully. During curriculum learning the width and height of the mazes are increased by four for each lesson.

All of the visible child objects that are in the learning environment objects in the `Controller` are illustrated in Figure 3.2. Here are more details descriptions for them:

1. The `MazeAgent` is the `GameObject` that houses the `Agent` script along with

the sensors that it can use.

2. The background is a plane that serves to provide a cleaner background for the mazes and makes it easier to see the maze. As the concurrent learning environments are stacked vertically, the background also serves to block the view of the other learning environments. Reason for stacking the environments vertically is to prevent the `Raycast Perception Sensor` that are in the `Agents` from detecting the other environments.

3. Goals `GameObject` serves as a container for all of the possible goals that are in the maze. `Goals` object is filled whenever a new maze is generated or whenever a `Goal` has been reached. The `Goal GameObjects` are green cubes that fill a single tile in the maze almost completely.

4. The maze is generated by first creating the layout of the maze into a two-dimensional integer array. The algorithm used for the creation of the maze is not named by the tutorial [30]. After the layout of the maze is generated it is used to create a single 3D mesh that is the complete maze.

5. Start position is an invisible `GameObject` that is generated to the bottom left of the maze whenever a maze is generated. It serves as the store of the location to which the `MazeAgent` is sent when a new episode begins.

The hyperparameters that are used are the same for all of the `Agents` but due to the fact that the ML-Agents toolkit would automatically use curriculum learning if it is present in the configuration file, the curriculum learning `Agents` had their own configuration file. The hyperparameters used for this study can be found in Table 3.1. The curriculum learning's lesson plan is stored in the environmental parameter section as a subsection which is called curriculum.

The evaluation environment is made out of four mazes, one for each of the differently trained agents. These mazes are identical so that there cannot exist a

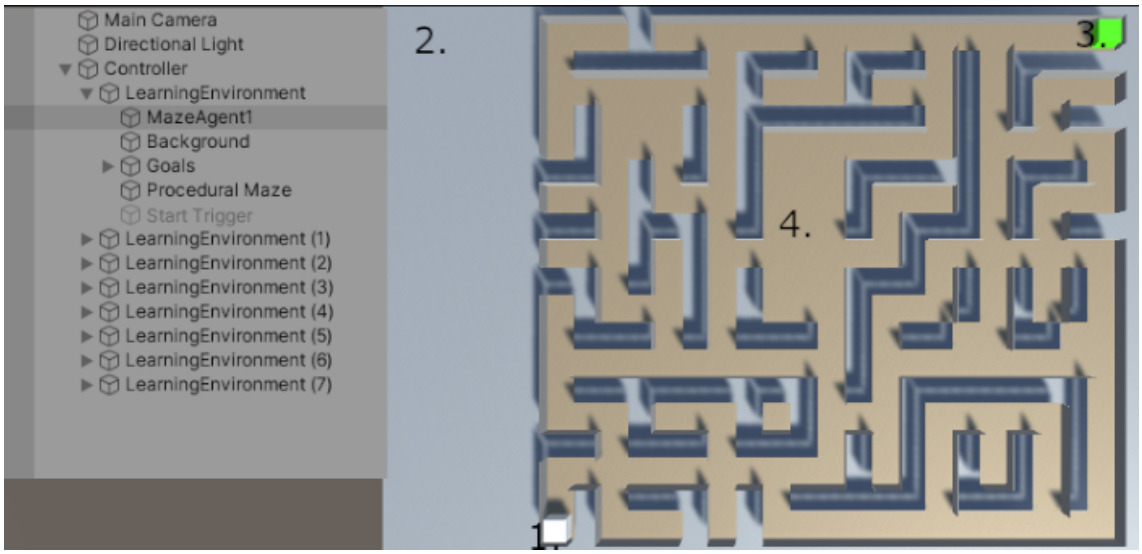

Figure 3.2: Illustration of the learning environment and the editor hierarchy of the Unity scene. 1) The MazeAgent that is currently at the start position; 2) The background; 3) The Goal; 4) The maze

possibility of a simple maze being generated for one `Agent` while creating a complex maze for any other `Agent`. The goal of the evaluation maze is the same as in the learning environment.

Each agent is given one minute to go through the maze once after which the time it took for the agent to finish is recorded. This kind of evaluation environment allows each `Agent` enough time to find their way through the maze as if the `Agent` could not complete the maze within a minute it is unlikely to clear it at all. Going through the maze once is enough for the evaluation as that shows that the `Agent` can navigate through a maze. After all four agents have finished or the time has run out the evaluation is restarted. These times are then written into a CSV file called ExaminationTimes.

Table 3.1: Sample of the important hyperparameters that are used in this paper

Hyperparameter	*Value*
Batch Size	100
Buffer Size	1000
Learning Rate	$1.0 * 10^{-5}$
Beta	$5.03 * 10^{-2}$
Epsilon	0.1
Lambd	0.95
Num Epoch	2
Learning Rate Schedule	Linear
Normalize	False
Hidden Units	128
Gamma	0.99
Strength	1.0

3.3 Training structure

The training consists of episodes that the `Agent` completes in order to improve its learning. These episodes consist of steps maximum amount of which can be specified in the Unity Editor. The `Academy` ticks these steps and the `Agent` chooses an action. How many steps are between the actions of the `Agent` depends on the settings set on the `Decision Requester` component.

In the beginning of an episode, the `Agent` is sent back to the start position which is located in the bottom left corner of the maze. From there the `Agent` starts to take actions in order to reach the goal. Once the goal is reached or a 1,000 `Academy` steps are taken the episode ends. When curriculum learning is used, the size of the maze is changed but the maze changes only when the generation of a new maze is called. This means that the `Agent` can have a maximum of four mazes that are of

a smaller size than what the curriculum would dictate.

The curriculum consists of four lessons. All but the last of these lessons have `Completion Criteria` as a subsection. The first lesson's `Completion Criteria` can be found in Table 4.2. For the second and third lesson's only differences are in the Threshold parameter in their `Completion Criteria` and the Value parameter of the lesson. The Threshold parameter increases by 0.1 in each lesson. The Value parameter is the height of the generated maze. In the first lesson the Value is 7 and it is increased by four for each lesson and in the final lesson the Value is 19.

Table 3.2: First lesson's `Completion Criteria` subsection

Parameter	*Value*
Measure	Reward
Behavior	MazeAgent
Signal Smoothing	True
Min Lesson Length	100
Threshold	0.2

4 Results

In this chapter I go through the data gathered from the training of the `Agent`s and the evaluations. Firstly, I go through how the data is gathered. After that, I go through the data that is divided into two case studies. The `Agent`s in these case studies use different sensors to gather observations from the surroundings.

4.1 Evaluation setup

The parts of training data that were collected for this study were:

1. The time it took to complete the training.
2. The amount of successful training sessions.
3. Average cumulative reward for each 10,000 steps.
4. Average entropy of the model for each 10,000 steps.

The time it took to complete was taken from a Python terminal after a training session was completed and then saved to a Google Sheets document. Also after each training session I also recorded whether it was a success. The success rate is given as a percentage value of the successes. The session was treated as a success, if the `Agent` managed to consistently reach cumulative reward values that were larger than zero for longer than 200,000 steps at the end of the training session. This also tells how well the `Agent` has managed to learn to explore the mazes. Both the

entropy and the cumulative reward were gathered from the model itself by using TensorBoard.

In order to see how well the usage of curriculum learning improves the learning of the agents, each agent runs through a series of training sessions. Each session gets stopped when the `Agent` completes 1.2 million steps. I chose such a large amount of steps due to the instability of each session. This should allow each `Agent` to have enough time for their training to stabilize. During these training sessions the ML-Agents toolkit saves many details of the training that can be used to analyze the training's quality. The sessions are categorized by the type of the sensor used by the `Agent` and whether or not that `Agent` used curriculum learning in its training.

As the computer I use to run the training sessions is the same computer that I use daily the training times have larger variance than if run with a dedicated computer. I reran any training which suddenly took significantly longer than the others. This usually happened when I had software running which put a lot of strain on my computer. The computer I use for the training of the `Agent` has a NVIDIA GeForce GTX 1060 6GB as a GPU, AMD Ryzen 7 1700X Eight-Core processor and 32GB of RAM. For the training, I set the inference device in the `Agent's Behavior Parameters` component to default which defaults to Burst. Burst is the newer method of using the computer's CPU for training. The other options are using the GPU or the legacy CPU inference device.

4.2 Results

4.2.1 Case Study 1

The first case study consists of the comparison of the results between using curriculum learning and not using it in the `Vector Sensor` using `Agent`s hereafter called Vector Agents. Neither of the Vector Agents managed to successfully complete all

of their training sessions but the Vector Agent using curriculum learning managed to complete significantly more of its training sessions successfully. With curriculum learning the success rate of training is 87.5% and without curriculum learning the success rate is 40%. The training of the Vector Agent without using curriculum learning took 48 minutes, 37 seconds and when using curriculum learning took 46 minutes, 37 seconds.

Vector Agent without curriculum learning

Most of the failures of the Vector Agent not using curriculum learning become apparent once at least 200,000 steps have been completed. If the Vector Agent's cumulative reward value hits zero at or before 200,000 steps, the Vector Agent can not recover and continues to gain no cumulative reward. However, a couple of times Vector Agent's cumulative reward hit zero after a period of otherwise successful training. Despite this these Vector Agents recovered from the period of zero cumulative reward and managed to complete the training successfully if the period was shorter than 200,000 steps.

The average cumulative reward of the Vector Agent without curriculum learning increases as the amount of steps increases when accounting the failed training sessions and continues to increase slightly until the end of the training. In the beginning the cumulative reward average is below 0.03 with a value of 0.026. This average cumulative reward looks quite stable but the reward value never increases above 0.1 even though during training the Vector Agent's cumulative reward values reached above 0.3 and even above and often are in between 0.2 and 0.3. This is due to the low success rate of training when not using curriculum learning. At the end of the training the average cumulative reward has a value of 0.08.

When the failed training sessions are not accounted for the average, the cumulative reward values are much more unstable and the increase of the cumulative

reward with increase of the steps taken is much more clear. At the beginning the average cumulative reward without failures is 0.045. When viewed this way the average cumulative reward becomes too unstable to see any clear increase in value after 600,000 steps and the average stays on both sides of 0.2. At the end of training the average cumulative reward is 0.208. Both of these averages can be found in Figure 4.1.

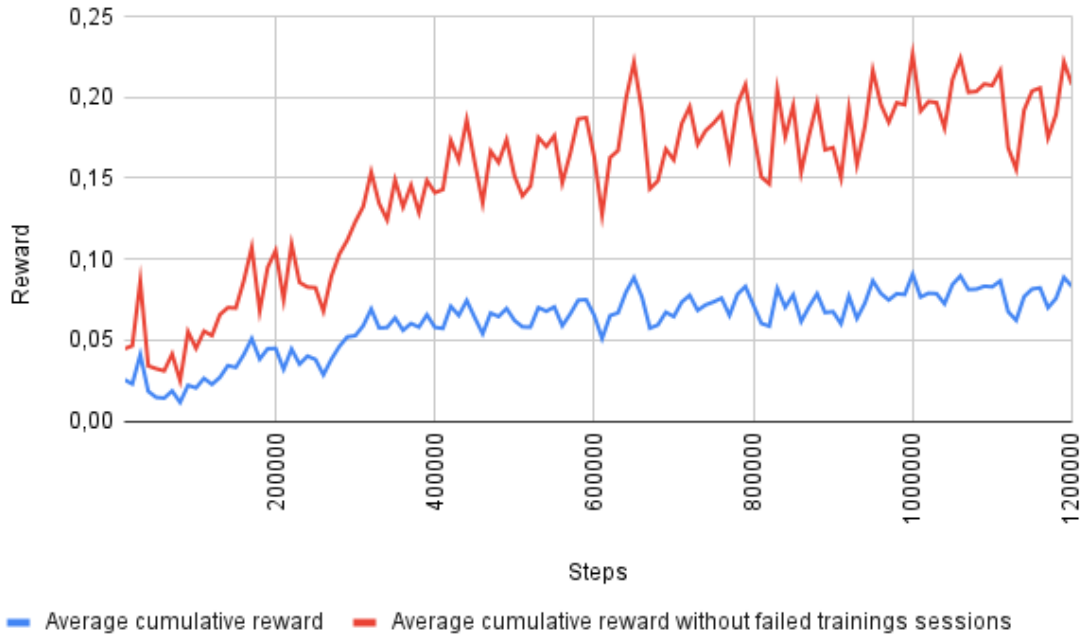

Figure 4.1: Average cumulative reward of the Vector Agents without curriculum learning with and without the failed training sessions

The average entropy for the Vector Agent without curriculum started at 1.450 and increased to 1.558 where it then stabilized at 100,000 steps with a slight downwards trend. From there the average entropy slowly decreases and at 600,000 steps the average entropy starts to decrease more rapidly. At the end of the training the average entropy has fallen to 1.481 with a clear downwards trend.

When the failed training sessions are removed from the average, the average

entropy starts lower than the total average and is 1.422. Then the average entropy rises and reaches its peak at 170,000 steps with a value of 1.558 but starts to decrease immediately after. This downwards trend continues until the end of the training and starts to decrease more rapidly at 960,000. At the end of the training the average entropy reaches 1.376. Both of these averages can be found in Figure 4.2.

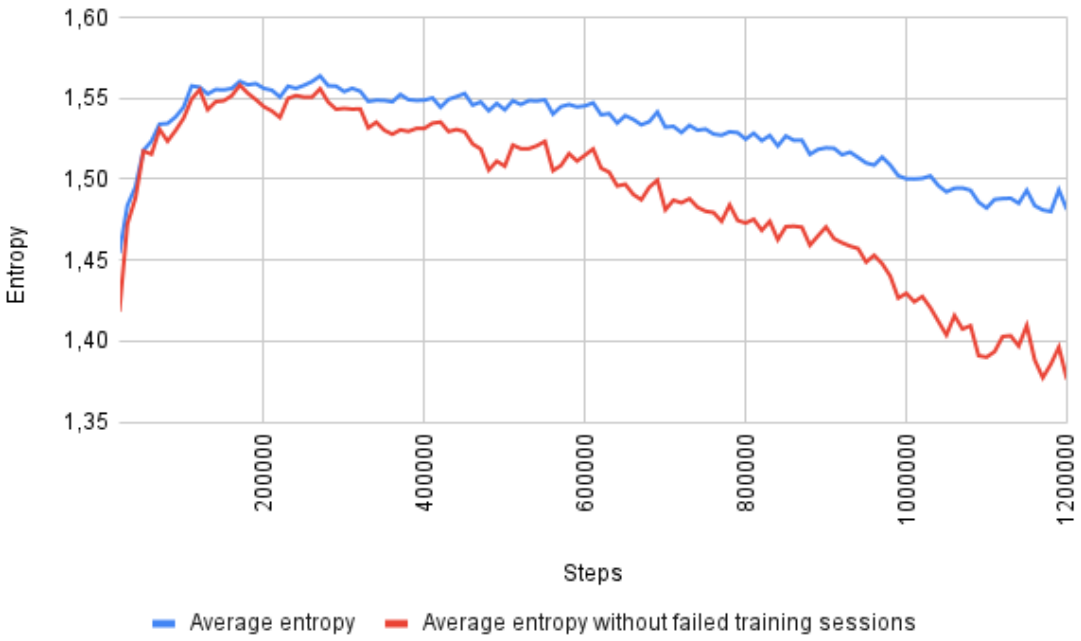

Figure 4.2: Average entropy of the Vector Agents without curriculum learning with and without the failed training sessions

Vector Agent with curriculum learning

For the Vector Agent using curriculum learning it is harder to spot the step count from which the training can be considered a failure as only five out of the forty training sessions were failures. After 320,000 steps every failed training session starts to have no cumulative reward and does not recover except for one successful training session where it managed to recover at 810,000 steps.

The average cumulative reward when using curriculum learning starts at just below 0.4 at a value of 0.3996 and starts to rapidly decrease due to the complexity of the maze increasing quickly and reaches its bottom at 150,000 steps with a value of 0.117. After this the average cumulative reward starts to increase and starts to stabilize to a value below 0.2 at around 400,000 steps. The average cumulative reward finished at a value of 0.173.

When the failed training sessions are not counted in the average, the average cumulative reward is similar to the total average but contains slightly higher reward values. At the start the average cumulative reward is at 0.401 and starts to decrease reaching the lowest point at 150,000 steps with a value of 0.111. From there the average cumulative reward starts to increase and fluctuates on the both sides of 0.2 peaking with a value of 0.253 at 870,000 steps. At the end of the training the average cumulative reward is just below 0.2 with the value of 0.197. These values can be seen in Figure 4.3.

The average entropy for the Vector Agent using curriculum learning starts at 1.543 and slowly rises reaching its peak at 260,000 with a value of 1.545. From that point the average entropy starts to decrease. At 680,000 the average entropy starts to increase again reaching its peak at 710,000 with a value of 1.474. After that the entropy continues to decrease reaching a value of 1.363 at the end.

Without the failed training sessions the average entropy follows closely the average entropy of all of the training sessions starting with a value of 1.541 and starts to increase. This increase reaches its peak at 180,000 with a value of 1.544 and stays at the same level until 260,000 steps are reached after which the average entropy starts to decrease. As with the average entropy of all of the training sessions the decrease in average entropy stops at 680,000 with a value of 1.456 and reaches its peak at 710,000 with a value of 1.461. Afterwards the average entropy continues to decrease and at the end reaches a value of 1.338. These values can be seen in Figure

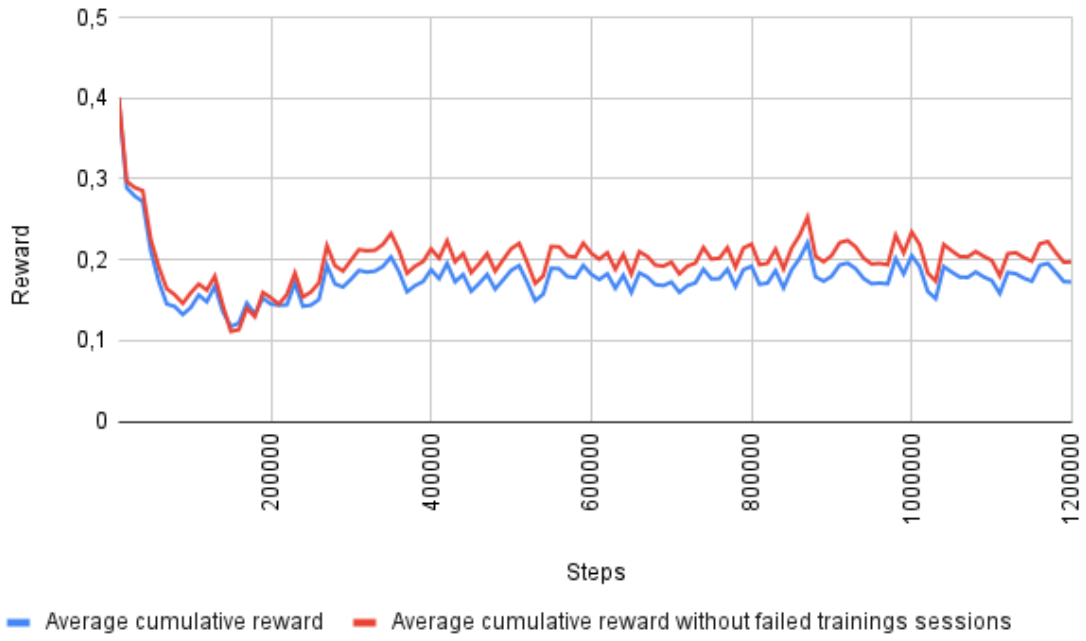

Figure 4.3: Average cumulative reward of the Vector Agents with curriculum learning with and without the failed training sessions.

4.4.

4.2.2 Case Study 2

The second case study consists of the comparison of `Raycast Perception Sensor` using `Agents` hereafter called Raycast Agents when curriculum learning was used and when it was not used. When curriculum learning is used the Raycast Agent completes all of its training sessions successfully while when not using curriculum learning results in some of the training sessions to be completed unsuccessfully. The resulting success rate when not using curriculum learning is 77.5%. The training of the Raycast Agent without using curriculum learning took 55 minutes, 53 seconds and when using curriculum learning took 56 minutes, 37 seconds.

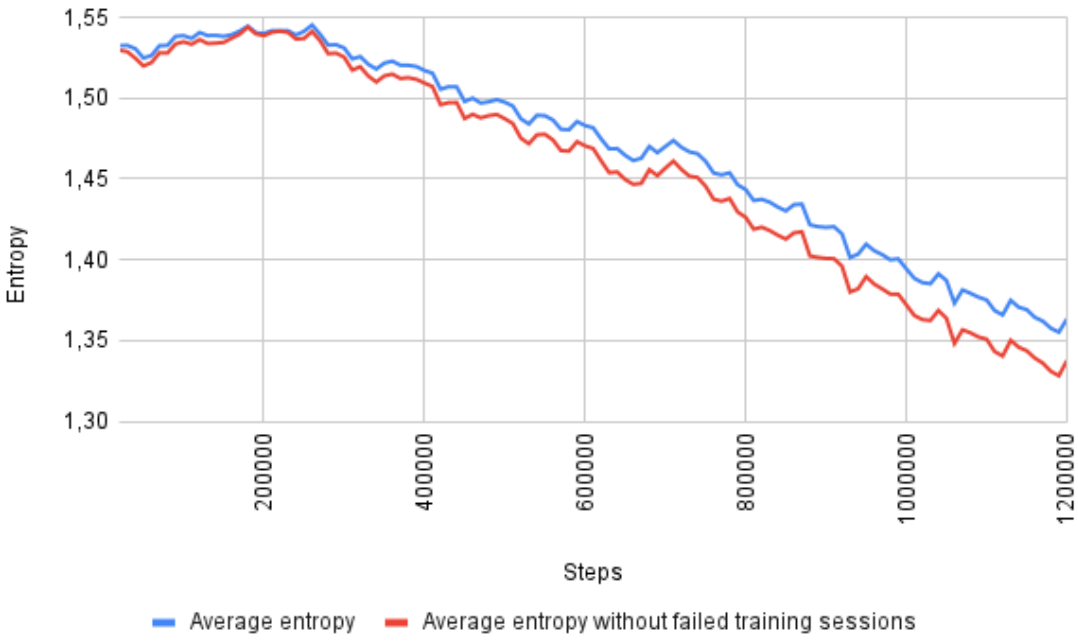

Figure 4.4: Average entropy of the Vector Agents with curriculum learning with and without the failed training sessions

Raycast Agent without curriculum learning

The point from which the Raycast Agent not using curriculum learning can no longer recover and succeed in its training is difficult to spot as many successful training sessions go for a long time with no cumulative reward. One of these sessions started to succeed in its training episodes after failing for 830,000 steps.

At the start of the training the average cumulative reward of all of the training sessions is 0.0008. The average cumulative reward then stays at really small values until at 100,000 steps with a value of 0.008 and after which the average cumulative reward starts to steadily increase until at 700,000 with a value of 0.210. After this point the average cumulative reward shows no clear increase and starts to become unstable fluctuating between 0.161 and 0.226 with a final value of 0.187.

When the failed training sessions are taken away from the average cumulative

reward starts at 0.001. The average cumulative reward stays low until at 100,000 with a value of 0.010 when the average cumulative reward starts to increase. This increase stops at 700,000 steps when it has a value of 0.271 and the average cumulative reward starts to become unstable and stays as such until the end. During this period the average cumulative reward fluctuates between 0.207 and 0.292. In the end the average cumulative reward reaches a value of 0.242. These values can be seen in Figure 4.5.

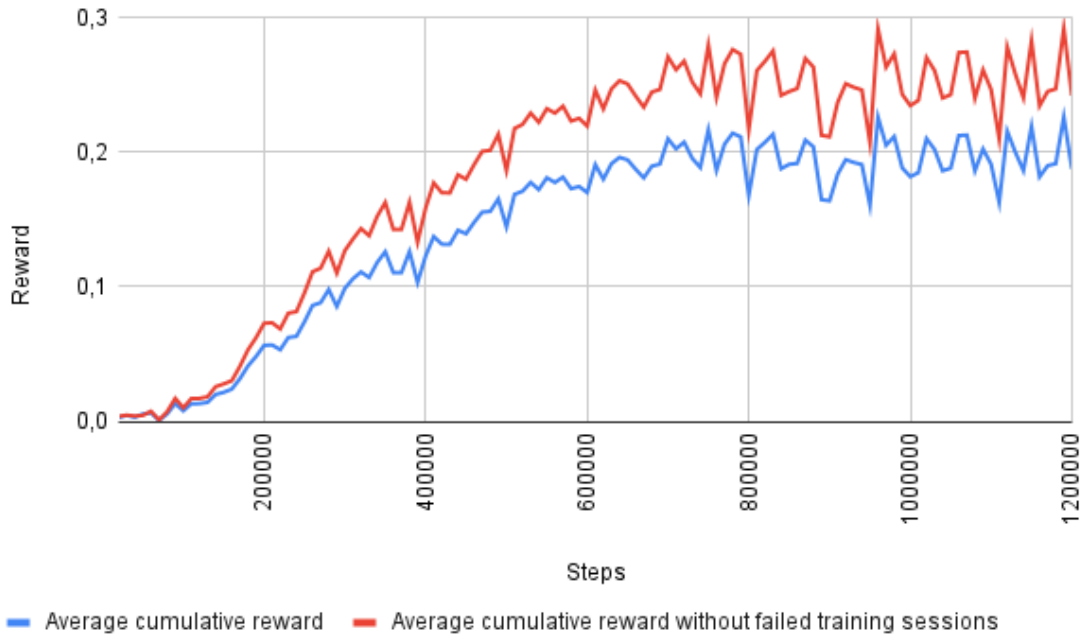

Figure 4.5: Average cumulative reward of the Raycast Agents with curriculum learning

The average entropy of the Raycast Agent not using curriculum learning with all of the training sessions starts at 1.609 and starts to slightly decrease. The decrease of the average entropy increases at 190,000 steps with a value of 1.598. Once 800,000 steps are reached the average entropy starts to decrease slower. At this point the average entropy is 1.483. The decrease of average entropy continues to slow down

until the end. The final value of the average entropy of all of the training sessions is 1.461.

When the failed training sessions are removed from the average entropy it starts at 1.609. It then starts to slowly decrease. At 190,000 steps with a value of 1.600 the average entropy starts to decrease faster and continues to decrease with this new rate until 800,000 steps when the average entropy reaches a value of 1.456. This slower descent of the average entropy continues to the end. The final value of the average entropy without failed training sessions is 1.428. Both of these can be seen in Figure 4.6.

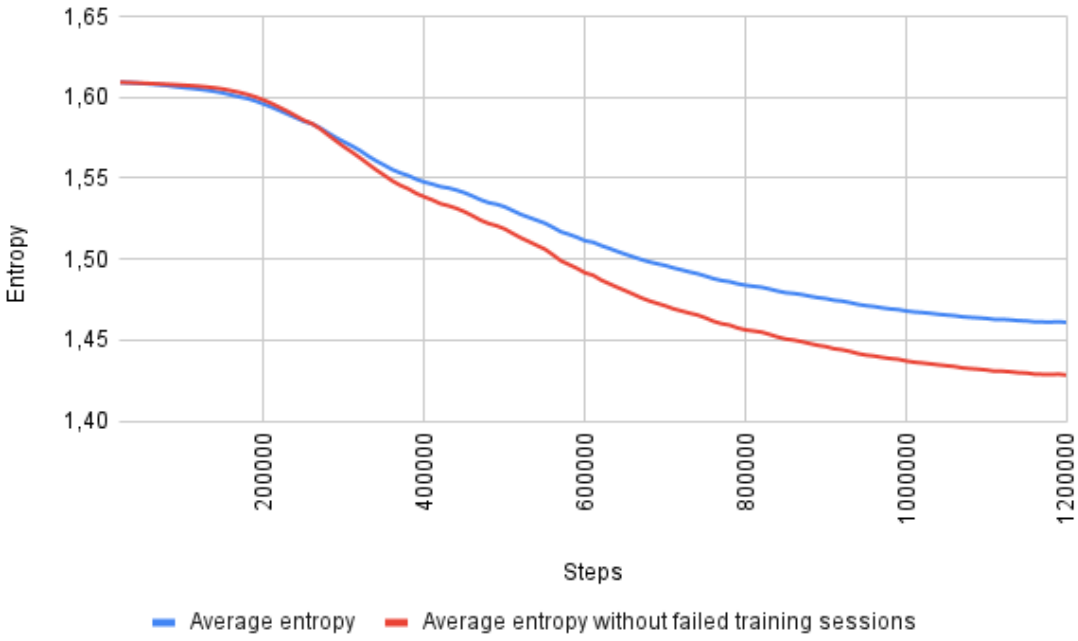

Figure 4.6: Average entropy of the Raycast Agents with curriculum learning

Raycast Agent with curriculum learning

When using curriculum learning in the training of Raycast Agent the average cumulative reward starts with a high value of 0.566 but starts to decrease shortly after

the beginning reaching its lowest value at 50,000 steps with a value of 0.210. After this the average cumulative reward varies until at 140,000 when the value reaches 0.361 and starts to then decrease in a stable manner. At 230,000 steps the decrease of the average cumulative rewards stops at a value of 0.244. After this point the average cumulative reward stay stable until after 100,000 steps after which the average cumulative reward shows greater variance reaching values between 0.222 and 0.287. At the end of the training the average cumulative reward is at 0.250. This can be found in Figure 4.7.

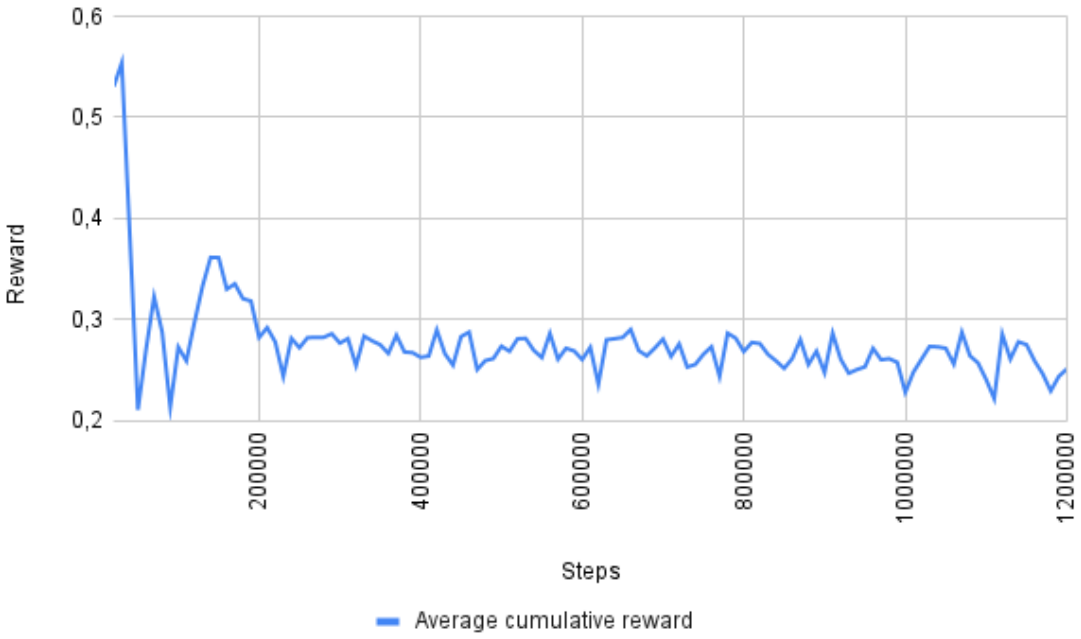

Figure 4.7: Average cumulative reward of the Raycast Agents with curriculum learning

The average entropy of the Raycast Agent using curriculum learning starts at 1.609 and starts to decrease at a slow pace until 80,000 steps with a value of 1.604 after which the average entropy starts. The decrease in entropy starts to slow down at 330,000 steps with a value of 1.471. After this point the average entropy starts to

slightly increase reaching a value of 1.465 at 580,00 steps. Once this small increase has stopped the average entropy continues to decrease until the end reaching a value of 1.438. This can be seen in Figure 4.8

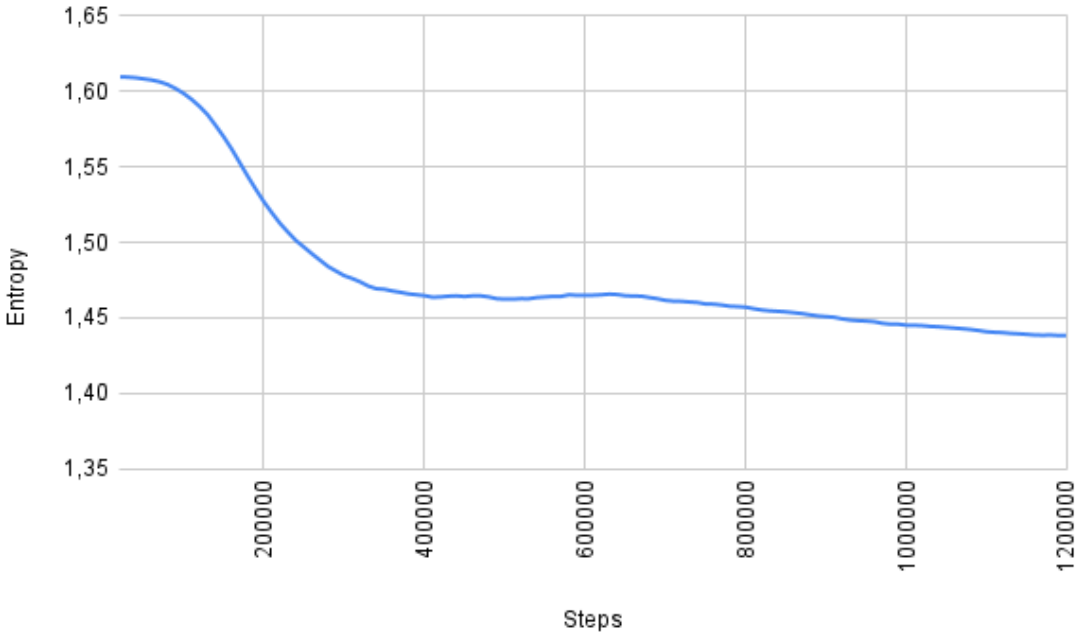

Figure 4.8: Average entropy of the Raycast Agents with curriculum learning

4.2.3 Evaluation results

The criteria for an `Agent` to be chosen for the evaluation is that it has the lowest entropy of the successful training sessions. The evaluation of the `Agent`s is repeated for a total of two hundred times and out of the 40 training sessions completed for each `Agent` the `Agent` with the lowest entropy value at the end of training and has completed the training successfully is selected to complete the evaluation. From the evaluation three data points are collected from the four different `Agent` variations. These data points are:

1. Average completion time of the evaluation.

2. Average completion time of the evaluation without the failed evaluations counted.

3. Success rate of the evaluation.

These data points are collected from a CSV-file created in the evaluation environment. The CSV-file contains the time it took for each `Agent` to complete the evaluation and the number of the round. The success rate is calculated from the completion times by dividing the successful evaluations with the total evaluation count. Evaluation is counted as a success if the time it took to complete is less than one minute.

The success rate for the Vector Agent when not using curriculum learning is 31.5% and when curriculum learning is used the success rate is 22.5%. For the Raycast Agent the success rate when not using curriculum learning is 52% and when the curriculum learning is used the success rate is 53.5%.

The average completion time of the evaluation for the Vector Agent when not using curriculum learning is 47.07 seconds and when the failed evaluations are taken out of the average the time is 18.97 seconds. When curriculum learning is used with the Vector Agent the average completion time of the evaluation is 49.94 seconds and when the failed evaluations are taken out the average completion time is 15.30 seconds.

For the Raycast Agent when not using curriculum learning the average completion time of the evaluation is 40.39 seconds and when the failed evaluations are taken out of the average the completion time is 22.29 seconds. When curriculum learning is used the average completion time of the evaluation is 38.89 seconds and when the failed evaluations are taken out of the average the completion time is 20.53 seconds.

4.3 Summary

The training sessions for Vector Agent that does not use curriculum learning takes on average 48 minutes, 37 seconds and has a success rate of training is 40%. For the Vector Agent that uses curriculum learning, the training sessions take on average 46 minutes, 37 seconds and the success rate of training is 87.5%. The Raycast Agent that does not use curriculum learning takes on average 55 minutes, 53 seconds to complete its training sessions and has a 77.5% success rate of training. For the Raycast Agent that uses curriculum learning, the training sessions take on average 56 minutes, 37 seconds and it has a success rate of training of 100%. These can also be seen in Table 4.1 and Table 4.2.

Table 4.1: Raycast Agent data gathered from the evaluation

Type	*Raycast*	*Raycast Curriculum*
Training times	55 minutes, 53 seconds	56 minutes, 37 seconds
Training success rate	77.5%	100%
Evaluation times	22.29 seconds	20.53 seconds
Evaluation success rate	52%	53.5%

Table 4.2: Vector Agent data gathered from the evaluation

Type	*Vector*	*Vector Curriculum*
Training times	48 minutes, 37 seconds	46 minutes, 37 seconds
Training success rate	40%	87.5%
Evaluation times	18.97 seconds	15.30 seconds
Evaluation success rate	31.5%	22.5%

In the first case study, the average cumulative reward without the failed training sessions for the Vector Agent that does not use curriculum learning starts at just below 0.05 with a value of 0.045 increasing from there, and at the end of the training

the value is 0.208. The average entropy without the failed training sessions for the Vector Agent that does not use curriculum learning starts at 1.421 and then quickly increases and after the value has started to plateau starts to decrease and ends with a value of 1.376.

For the Vector Agent that uses curriculum learning, the average cumulative reward without the failed training sessions starts at a value of 0.401, and at the end of the training, the average cumulative reward has a value of 0.197. The average entropy without the failed training sessions for the Vector Agent that uses curriculum learning starts at 1.541 and ends with a value of 1.338.

In the second case study, the average cumulative reward for the Raycast Agent that does not use curriculum learning starts at a value of 0.0008 and reaches a value of 0.242 at the end of the training. For the average entropy, the start value is 1.609 and after a smooth decrease the average entropy's final value is 1.428.

The Raycast Agent that uses curriculum learning has a value of 0.566 for the average cumulative reward at the beginning and then decreases really quickly and after 230,000 steps the decrease ends. The final value of the average cumulative reward is 0.250. The average entropy of the Raycast Agent that uses curriculum learning starts at a value of 1.609 and starts to quickly decrease and at around 330,000 steps the decrease starts to slow down. In the end the value of average entropy is 1.438.

In the evaluation, the Vector Agent that does not use curriculum learning has an average completion time of the successful evaluations of 18.97 seconds and completes 31.5% of the evaluations. The Vector Agent that uses curriculum learning completes the successful evaluations on average in 15.30 seconds and has a success rate of 22.5% of the evaluations. The Raycast Agent that does not use curriculum learning completes the successful evaluations on average in 22.29 seconds and completes 52% of the evaluations. The Raycast Agent that uses curriculum learning has an average

completion time of 20.53 seconds for the successful evaluations and completes 53.5% of the evaluations successfully. These can also be seen in Table 4.1 and Table 4.2.

5 Analysis

In this chapter, I compare and analyze the results gained in both of the case studies. The two case studies are not compared to each other. When comparing the results of the successful training sessions only a subset of the results are taken. This subset is the size of successful training sessions of the `Agent` not using curriculum learning. This is done in order to make the results more comparable. From the `Agent` that are using curriculum learning the training sessions that are removed are the ones that have the lowest cumulative reward at the end of the training. From the remaining training sessions both the cumulative reward and entropy are collected and then put into an average.

The results from both of the case studies show that implementing an intentionally unoptimized curriculum learning in the training stabilized the training significantly and increased the success rate the `Agent` has in the evaluation and in training. The curriculum used in this study is taken almost completely from the documentations of the ML-Agents toolkit. Only the threshold values and the measure are changed from the original example provided in the documentation.

In both case studies, the average cumulative reward started to become unstable after 600,000 steps. In this context, unstable means that the difference between nearby values is high. This can be more clearly seen when the failed training sessions are removed. Even in the case of the Raycast Agent with curriculum learning the average cumulative reward starts to fluctuate more after the 600,000 steps mark.

This indicates that 600,000 steps is a good cutoff point for the training of these `Agent`s as continuing the training for any longer will only slightly improve the results.

5.1 Case Study 1

5.1.1 Training results

In the first case study, the Vector Agent without curriculum learning the average cumulative reward stays below 0.1 through the training due to the large number of failed training sessions but it seems to try and reach 0.1. When the failed training sessions are removed from the average the cumulative reward values start to reach a value of 0.2 and are quite unstable towards the end of the training. This is likely due to the prevalence of cumulative reward values of zeros that are in the successful training sessions of the Vector Agent. This makes the early parts of the average look more stable.

When viewing the average cumulative reward for the Vector Agent not using curriculum learning the value of the average continues to increase right up to the end of the training. This indicates that the average cumulative reward has yet to completely stabilize but as the increase is quite slow and unstable, it seems unlikely that prolonging the training any more would provide much better results.

When curriculum learning is used the average cumulative starts to decrease right from the start and the reason for this behavior is due to the way rewards are given to the `Agent`. As stated in Chapter 3 the reward is calculated by dividing the maximum steps for an episode with the amount of steps taken at that point. This results in the `Agent` gaining higher rewards at the beginning as the maze is smaller and simpler.

The average cumulative reward's value with the Vector Agent using curriculum learning stabilizes at around 400,000 at around a value of 0.2. This indicates that

when using curriculum learning the Vector Agent reaches its maximum reward value much earlier than when not using curriculum learning. This is in accordance with one of the hypotheses given by Bengio et al. [18], which states that the training of the machine learning model would be faster when using curriculum learning.

Unlike when curriculum learning was not used, the value of average cumulative reward starts to become unstable after 800,000 steps and not after 600,000 steps. This also strengthens the hypothesis that using curriculum learning makes the training more stable.

As stated earlier, the set of results for the Vector Agent using curriculum learning is reduced to be the same size as with the version of the Agent not using curriculum learning. In this case, the size of the subset of samples is 16. When comparing the average cumulative reward of both of the Vector Agents, the comparison shows that when curriculum learning is implemented in the Vector Agent the average cumulative reward values are consistently above the Vector Agent not using curriculum learning. This indicates that curriculum learning even in an unoptimized state can increase the cumulative reward that an `Agent` receives during training.

With only a subset of the successful training sessions of the Vector Agent using curriculum learning, they look more unstable at the end than with the Vector Agent not using curriculum learning. But as with the whole set of successful training sessions, the instability of the average cumulative reward of the Vector Agent using curriculum learning starts to become apparent only after 800,000 steps. The averages for both sets of the average cumulative reward can be seen in Figure 5.1.

When looking at the average entropy of the Vector Agent not using curriculum learning, it starts at a value that is almost lower than where it ends up in. This behavior where the entropy starts at a lower value and then increases happens with almost all of the training sessions and the reason for its appearance is unclear. I can only speculate for the reason of this kind of behavior and finding out why it

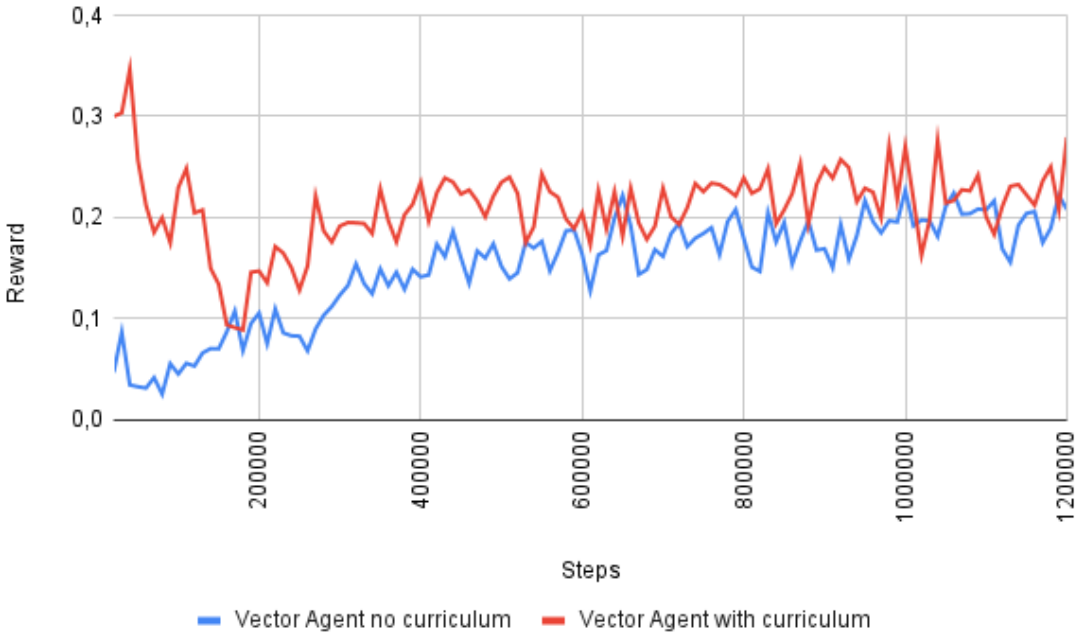

Figure 5.1: Average cumulative reward for both of the Vector Agents with a sample size of 16.

happened is left for further studies. Most likely the reason for this kind of behavior is in some way due to the simplicity of the Vector Agent that only happens when it is put into the maximum sized training maze.

In the average entropy for the Vector Agent not using curriculum learning, there is no clear sign of entropy becoming more unstable. This might be because the average entropy is somewhat unstable for the whole duration of the training and as such more instability is difficult to notice. This is shown by the average entropy's downward trend is not smooth meaning the `Agent` is reducing the randomness of its actions more often than it is increasing it. Also the average entropy stays quite high meaning in the end of the training the `Agent` is not sure which action is correct and still chooses options quite randomly.

When curriculum learning is used a clear downwards trend is seen in the aver-

age entropy and this time the average entropy becomes clearly more unstable after 700,000 steps. It even starts to go upwards before 600,000 steps are reached. The reason for this is a large amount of training sessions which have suddenly increased their entropy around that point. The reason for this is unclear but it might be a result of a series of mazes that were too difficult or complex for the Vector Agent to successfully complete.

When comparing both of the Vector Agents' average entropy, the Vector Agent using curriculum learning has its average entropy almost for all of the training lower than with the Vector Agent not using curriculum learning. This indicates that when curriculum learning is used the Vector Agent's actions are less random but only slightly as the difference in the values is less than 0.1. The averages for both sets of the average entropy can be seen in Figure 5.2.

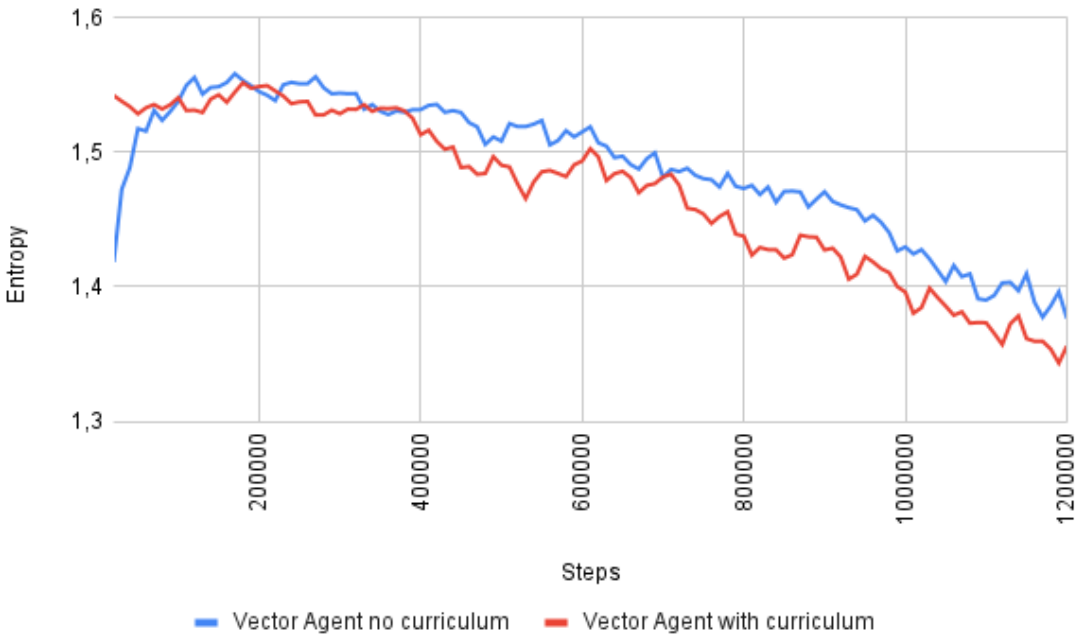

Figure 5.2: Average entropy for both of the Vector Agents with a sample size of 16.

Even though the difference in average cumulative reward and entropy are low

between the two Vector Agents, the Vector Agent which used curriculum learning managed to have a significantly higher success rate of training. It has a success rate of 87.5% and when curriculum learning is not used the success rate is 40%. This indicates that curriculum learning can increase the probability of a training session ending with an `Agent` learning a functional strategy even if the `Agent` does not appear to fare better in the training data.

5.1.2 Evaluation

The Vector Agent that uses curriculum learning has reached better results in both average cumulative reward and average entropy than the Vector Agent that does not use curriculum learning. This indicates that when curriculum learning is used the evaluation results will be better than when curriculum learning is not used although the training data from both Vector Agents are unstable which makes each training session's result in the evaluation hard to predict.

In the evaluation of the training results, the Vector Agent which used curriculum learning has a success rate of 22.5% and when curriculum learning is not used the success rate is 31.5%. When compared to the increase of the success rate of training, the Vector Agent using curriculum learning actually fared worse in terms of the success rate of the evaluation. This could indicate that the introduction of curriculum learning to the Vector Agent actually worsens the performance of the `Agent` but this can also indicate that using entropy as a sole factor for deciding which `Agent` to use can lead to the utilization of a subpar `Agent`.

In terms of the average completion time of the evaluation from total, the Vector Agent not using curriculum learning managed to clear the evaluation almost three seconds faster than the Vector Agent using curriculum learning but when the failed evaluations are taken out of the total the Vector Agent using curriculum learning is a little over three and a half seconds faster.

In order to understand this behavior I firstly compare only the 45 best from both of the Vector Agents. This amount is chosen because that is the amount of successful evaluations that the Vector Agent using curriculum learning has. With this limitation set the Vector Agent which does not use curriculum learning has an average completion time of 9.72 seconds and when curriculum learning is used the average completion time stays the same.

This leaves a question regarding the evaluation results: How does the Vector Agent that does not use curriculum learning have better total average completion time but lower average completion time without failed evaluations? In order to answer them I collect two additional data points from the evaluation data. These data points are:

1. Amount of evaluations that are under 15 seconds

2. Amount of successful evaluations that are over 15 seconds

The Vector Agent not using curriculum learning has 36 evaluations that are under 15 seconds and with the Vector Agent using curriculum learning the amount is 29. This is the reason why the Vector Agent not using curriculum learning has almost six seconds faster average evaluation completion time than the Vector Agent using curriculum learning as 80% of the evaluations used in the equalized comparison can be considered as fast completions of the evaluation for the Vector Agent not using curriculum learning.

On the other hand, out of all the successful evaluations the Vector Agent that uses curriculum learning has only 16 evaluations that are over 15 seconds. This comes out as this Vector Agent has 35.6% of the successful evaluations that are longer than 15 seconds. The Vector Agent not using curriculum learning has 26 evaluations that are completed in over 15 seconds. This comes out as the Vector Agent not using curriculum learning has 42.9% of the successful evaluations that are longer

than 15 seconds. This means while the Vector Agent that does not use curriculum learning has more quickly completed evaluations these make up a smaller part of the successful evaluations than the Vector Agent that uses curriculum learning indicating that this specific Vector Agent that does not use curriculum learning is better suited for a larger variety of mazes while this specific Vector Agent that uses curriculum learning is on average better at the mazes it has completed.

5.2 Case Study 2

5.2.1 Training results

The average cumulative reward for the Raycast Agent starts at almost zero but then starts a clear climb upwards with the climb plateauing to around 0.25. After 600,000 steps the values of the average cumulative reward starts to become more and more unstable and after 800,000 steps the average starts to reach maximum instability. This reinforces the notion that a good stopping point for the training of these `Agent`s is at around 600,000 steps and at latest 800,000 steps. This instability is much clearer when the failed training sessions are removed from the average.

As there are no failed training sessions for the Raycast Agent that uses curriculum learning, it continues to reinforce the result that introduction of curriculum learning will stabilize the training results. The quick drop in the average cumulative reward at the beginning is due to the way rewards are given to the `Agent` as it gives much higher rewards when a maze is smaller. The instability that is evident in the average cumulative reward before 200,000 steps are reached is due to the different `Agent`s having reached a different part of the curriculum. After 200,000 steps, most of the training sessions have reached the final lesson of the curriculum and at 300,000 steps every training session has reached the final lesson.

The average cumulative reward of the Raycast Agent that uses curriculum learn-

ing starts to plateau at around 300,000 steps around a reward value of 0.25. Slightly after 400,000 steps are reached there is a period of instability after which the average cumulative reward values start to stabilize. The average starts to slightly destabilize again after 600,000 steps are reached with the instability becoming clearer after 800,000 steps and becoming even stronger once 1,000,000 steps are reached. This indicates a good cutoff point for the training of somewhere between 400,000 and 600,000 steps. Continuing the training after 600,000 steps are reached would only serve to destabilize the training while not providing any increase in the cumulative reward values that the Raycast Agent would gain.

Changing the cutoff point would also require more information than just what is available with this will require optimization of the training's hyperparameters as the entropy of the `Agent` might not reduce at an acceptable pace in a shorter training making the `Agent`s actions too random.

When comparing the two Raycast Agents' average cumulative rewards the Raycast Agent that uses curriculum learning has its successful training sessions reduced to 31. In the comparison it becomes clear that the Raycast Agent that uses curriculum learning plateaus at around 0.25 almost 400,000 steps before the Raycast Agent that does not use curriculum learning but on the other hand the Raycast Agent that uses curriculum learning has more instability in the earlier portions of the training. This indicates that instability starts to occur in an increasing manner after a certain amount of steps from which the cumulative rewards start to plateau. The graph with both of the Raycast Agents' average cumulative reward compared can be found in Figure 5.3.

The average entropy of the Raycast Agent that does not use curriculum learning shows no signs of instability and after a plateauing until at around 200,000 steps it starts a steady decrease right up until the end of training where it once again starts to plateau. As also seen in the first case study it seems that even though the

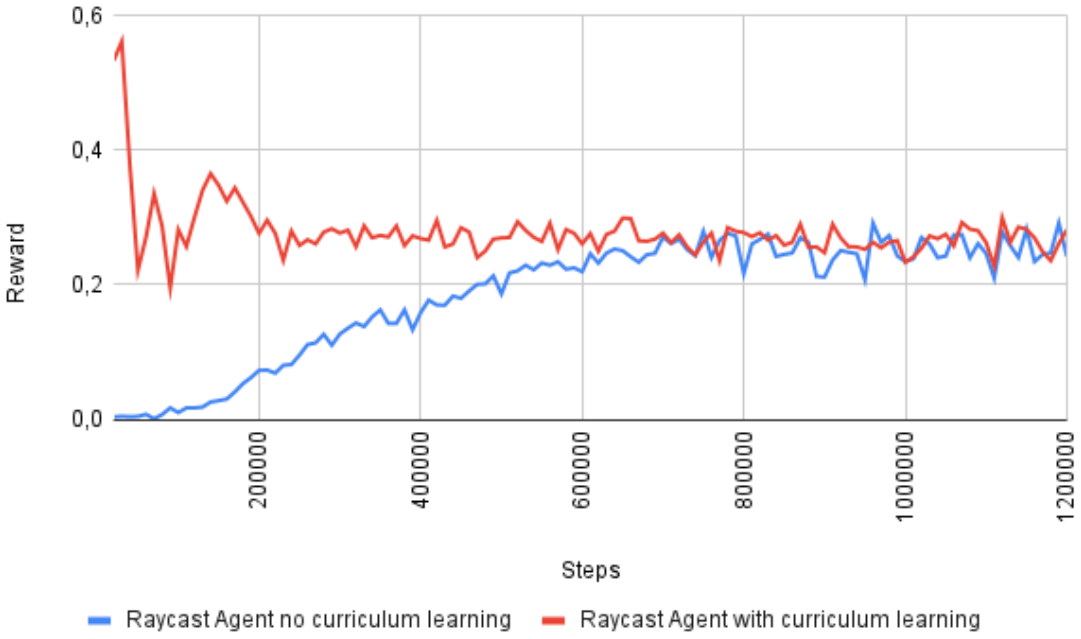

Figure 5.3: Average cumulative reward for both of the Raycast Agents with a sample size of 31.

average cumulative reward starts to become unstable after a certain point, no such behavior is to be seen in the average entropy.

For the Raycast Agent that uses curriculum learning, the average entropy starts to decrease quickly in the beginning and then at around 400,000 steps the average entropy starts to plateau with no significant changes until after 600,000 steps. At that point the average entropy starts to once again decrease but this time slowly. This reinforces the notion that the cutoff point for training is somewhere between 400,000 and 600,000 steps.

When comparing the two Raycast Agents' average entropy, the size of the used subset is 31. With this it is clear to see the difference in average entropy. While the Raycast Agent that uses curriculum learning reaches its plateau much earlier both of their average entropy values reach around the same value in the end. This value

is still quite high meaning the Raycast Agents have a high randomness of actions. This comparison can be seen in Figure 5.4. With these results it is clear that in case of the Raycast Agent, using curriculum learning makes the cumulative reward gain more stable and makes the training take significantly less time.

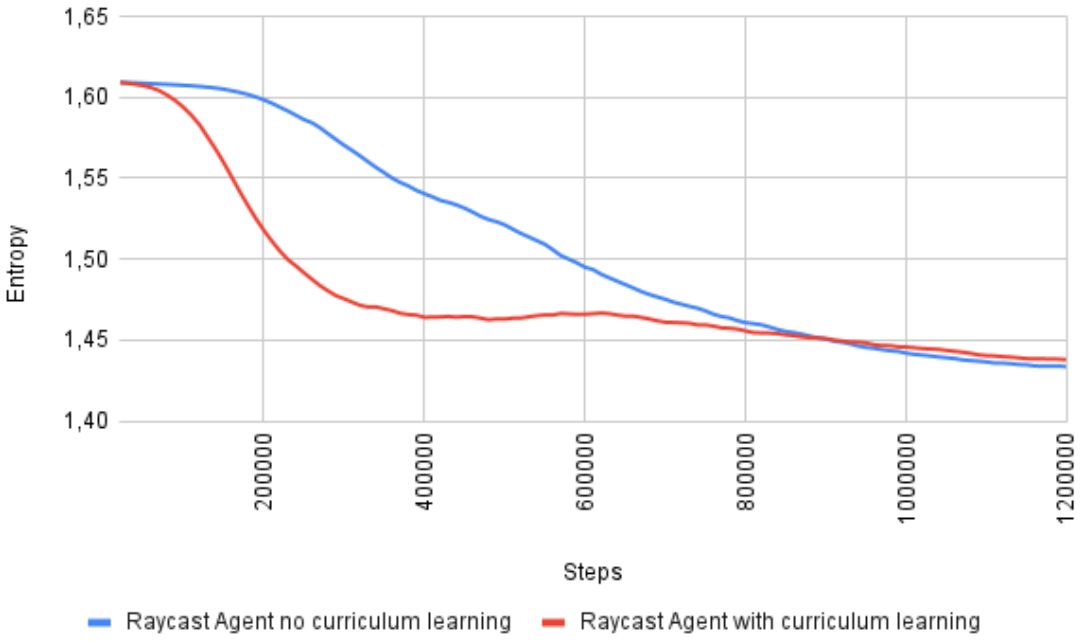

Figure 5.4: Average entropy for both of the Raycast Agents with a sample size of 31.

5.2.2 Evaluation

Overall the difference between the training data of both of the Raycast Agents is greatly different shortly after the beginning of the training but when the training starts to reach its end both the average cumulative reward and average entropy start to converge to the same point meaning both of the Raycast Agents should have quite similar evaluation results.

In the evaluation of the training results of the Raycast Agents, the Raycast Agent

that uses curriculum learning has a success rate of 53.5% and the Raycast Agent that does not use curriculum learning has a success rate of 52%. As can be clearly seen with these success rates, there is not much of a difference between the two with a 1.5 percentage point difference in favor for the Raycast Agent that uses curriculum learning.

This small difference continues with the average completion times of the evaluations as the difference in total average completion times is only 1.51 seconds in favor of the Raycast Agent that uses curriculum learning. When the failed evaluations are taken out from completion time average the difference is 1.76 seconds in the same direction as before.

When the amount of successful evaluations is made to similar sizes, the difference in completion times becomes bigger but still stays quite small. In this situation the Raycast Agent that does not use curriculum learning has the same completion time as when the failed evaluations are removed and the Raycast Agent that uses curriculum learning has a completion time of 19.49 seconds. This makes the difference between the completion times to 2.81 seconds.

In light of this data, it is clear that despite the fact that curriculum learning made the Raycast Agent succeed in all of the training sessions it did not make the Raycast Agent better in a meaningful way in the evaluation. Then again this is to be expected when looking at the average cumulative reward and the average entropy of both of the Raycast Agents as both of those data points end up with nearly the same values. In this case the only advantage of using curriculum learning is that with it the training can be done in a shorter time.

5.3 Summary

The difference between using and not using curriculum learning is most clearly seen in the training data. In the first case study, the Vector Agent that uses curriculum

learning fares consistently better in terms of the training data indicating that curriculum learning even in an unoptimized state can make the training more stable. On the other hand, the Vector Agent that uses curriculum learning fared worse in the evaluation in almost every way although this is most likely due to the fact that both of the Vector Agents have high instability of their training.

In the second case study, the difference between the two types of Raycast Agents is less clear as both the average cumulative reward and average entropy for both of them start to converge on the same value. The main difference with the two Raycast Agents in results of the training is that the Raycast Agent that uses curriculum learning has its average cumulative reward and average entropy plateau much earlier than the Raycast Agent that does not use curriculum learning. This means that the usage of curriculum learning in this situation does not improve the quality of the training significantly but does on the other hand shorten the time needed for training. For the evaluation, both Raycast Agents fared similarly and the Raycast Agent that uses curriculum learning fated only slightly better but this can also be attributed to the individual differences that appear in each training session.

6 Discussion

In this chapter, I discuss three main topics. Firstly I will discuss what this thesis proved and how it could be improved for further studies. Secondly I will discuss the Unity ML-Agents toolkit's usage in studies and video games. The final topic is about the different ways that machine learning could be used in video game development.

6.1 Postmortem

This thesis proved that curriculum learning even in an unoptimized state helped to stabilize the training and that it can improve the results of the trained `Agent`. Although in the case of the first case study, the results of the Vector Agent's training are too unstable to be able say with any certainty if using curriculum learning improves the evaluation. Even if using curriculum learning would always give worse results in the evaluation the improvements in training stability can still make implementing curriculum learning be worth it in certain scenarios.

The results of this thesis highlight the importance of having hyperparameters that are optimized for the specific scenario where it is used. Even though the results of this thesis were positive for unoptimized curriculum learning, the final results were still unstable in both of the case studies. Having optimized hyperparameters and curriculum learning would have made the end result of the training stable but that would have made the point of this thesis moot.

The fact that this study had two case studies that had different ways to gather

observations serves to show that curriculum learning, while it often improves the quality of the training, is not something that will guarantee that the improvements from it will be what were expected. That is why it is important to experiment more with curriculum learning so that best practices appear and use cases for it become clearer.

One way that the results of this thesis could have been improved was by creating a more complex environment. More complex in this context means that the environment has more way for the `Agent` to interact with it. For example there could have been certain tiles in the maze that stop the `Agent` for a certain amount of steps. While this thesis serves as to show the effects of curriculum learning, a more complex environment would have been a better example for someone who would want to use machine learning for an actual video game. There is only so much that one can learn from simple environments.

The effects of unoptimized curriculum learning would have been even better showcased with even more case studies. There are two more sensors in the Unity ML-Agents toolkit that could easily be added to the `Agent` and made into new case studies: `Camera Sensor` and `Grid Sensor`. The reason why these were not added to this thesis was time constraints. As stated in Chapter 3, the `Agent`s were trained on my only computer and the 40 training sessions that were done for one case study took on average little over 34 and a half hours. This means that for almost three days per one case study, I could not use my computer for almost anything else. But as the project with which the study of this thesis was made is available in GitHub, these additional case studies can be added later by anyone interested in studying this topic further.

6.2 Thoughts on Unity ML-Agents toolkit

The Unity ML-Agents toolkit is a good tool for those interested in studying machine learning. The documentation for it are quite comprehensive but there are some problems. One of these problems is that some parts of the documentation expect the reader to already understand the terminology of machine learning and the some other parts are written for someone whose just beginning to learn about machine learning. As an example, the documentation page that contains information on the hyperparameters [28] has parts of it written in a way that expects the reader to already know about the inner workings of their machine learning algorithm of choice.

If Unity ML-Agents toolkit is used in development of a product and the `Agent` that is trained fails its training session irregularly can make it hard to gather meaningful statistical data from the training. In this kind of situation, the entropy of the `Agent` often does not decrease but instead it actually increases. If the fix is not apparent and the lessons for the curriculum are easy to discover in the training environment, using curriculum learning can help to debug the training. With this I propose a topic for further studies: the usage of curriculum learning to assist in debugging the training of a machine learning agent.

The amount of studies done using Unity ML-Agents toolkit has been increasing over the years and will most likely to continue increasing as the toolkit improves, however there is a lack of studies that utilize curriculum learning. This along with the small amount of non-research related works for the toolkit made it difficult to find support for any issues that arose during the development of the environment and made it much more difficult to know how to analyze the results of the study.

Many of the studies that utilize Unity ML-Agents toolkit focus either on physics-based problems like teaching the agent to walk, or on simpler game environments. This makes it hard to utilize the toolkit in an actual game development setting. For

many researchers in the field of machine learning, this is fine as they use the video game environments to emulate the real world. But machine learning can be used to improve the development and quality of video games as well but this area of study is not as much studied.

For the usage of Unity ML-Agents toolkit for video game development, I am not sure if it will gain mainstream adoption in its current form. As I alluded to in the previous section, a clear set of best practices on how to optimize the hyperparameters has to be made. This along with the need to do a lot of manual changes to the hyperparameters also slow the adoption of the toolkit.

6.3 Usage of machine learning in video games

There are many different ways that machine learning could be utilized to improve the quality of video games. In my opinion, the two main ways are content creation and AI. For content creation machine learning could be utilized in a way to assist the art teams to create variant textures from a handmade example. One example of machine learning being used in the content creation of a video game is the game Source of Madness in which the enemy monsters are generated with procedural generation and are then animated using machine learning through the Unity ML-Agents toolkit [32]. In my opinion, with Source of Madness showing the way, the most likely usage for machine learning in video game content creation is as a tool that assists with procedural generation.

One way that machine learning could be used in video games is as a way to create replacements for more traditional algorithms. Just like how machine learning is used for data analysis in fields other than video games, it can be used and might already be used by the bigger video game developers and publishers for more traditional data analysis roles. One interesting application for machine learning in this regard is to use it as a way to balance matchmaking in online video games. There are

studies and projects that try to replace traditional matchmaking systems but these have not reached widespread adoption.

For the usage of machine learning as a way to create better AI in video games, the majority of studies focus on creating an AI opponent for the player. While this is an important area to improve in video games, especially in strategy games where the AI opponents are often making poor decision and cheat their way out of the games mechanics, there is one type of AI in video games that, in my opinion, could benefit more from the utilization of machine learning and it is AI companions. Many players are often frustrated by the companions that are in video games as the companions often provide little to no assistance.

On the other hand, a good companion AI must not outshine the player. This presents an interesting problem for the usage of machine learning as the `Agent` must be proficient AI but it cannot be better than the actual player. Of course this dynamic changes depending on the purpose of the companion AIs. For some games and players, the playable character might serve a supporting role, improving abilities for the companions. This might require that the `Agent` receives rewards from different actions than if the `Agent` served as a supporting companion.

No matter for which role the AI is used for, there is a problem that comes with using machine learning. This problem is that the machine learning agent tries to find the most optimal way to be the role it is trained to be. This is often the desired outcome for the agent but for a player playing against an AI that only makes optimal choices is not a desirable outcome. This problem can also appear with companion AIs with the companion taking optimal actions constantly making them outshine the player. In both of these scenarios most players would not consider the game as fun. One solution for this problem was explored by Sestini et a. [33] where they made the AI sometimes choose actions which were not the most optimal.

One way that results from studies of machine learning in video games could be

improved is by making the environments much more akin to an actual game. In my opinion, it is quite clear that the learning environment for a machine learning agent must be as close to the actual game environment as possible. If the agent does not use any kind of visual sensors for gathering observations the graphics of the environment do not need to be like they would be in the game but any tags or layers that could be used must be set as if they would be for the actual game.

I believe it will still take many years until machine learning has reached widespread adoption in the field of video games. One reason is that bigger video game developers and publishers avoid too big risks and implementing machine learning into a video game is a big risk. This is due to the lack of examples that are made outside of academia and research groups.

7 Conclusion

This thesis proved that using curriculum learning even in an unoptimized state will improve the stability of the training and can result in improvements of the quality of the results of the training. As the study in this thesis was built up from two case studies that provided different insights. Both case studies differed only by the way that the `Agent` received observations: In the first case study it was solely through the `Vector Sensor` and in the second it was through the `Raycast Perception Sensor`.

Both case studies had the `Agent` go through a maze that was changed after five training episodes had ended. These training episodes continued until 1,000 steps were reached or the `Agent` had reached the goal. In both case studies the result of the `Agent` were compared against the same type of `Agent` that had curriculum learning implemented. The way that the curriculum was created was to make the maze start as a small and simple maze and as the `Agent` learns to complete these simpler mazes it is then made bigger and more complex.

In the evaluation of the training results each `Agent` had the same maze to complete. Each repeat of the evaluation ran until each `Agent` had completed it or until one minute had passed.The selection criteria of the `Agent`s for the evaluation was the lowest entropy at the end of the training and that the `Agent` had actually completed its training successfully. The evaluation was repeated 200 times and for each repeat, the maze was generated again.

In terms of the hyperparameters, there were no differences between them in both

of the case studies. The only difference in the configuration file with both the `Agent` that used curriculum learning and the `Agent` that did not use curriculum learning was the addition of the environmental parameters for using curriculum learning. The hyperparameters and the environmental parameters were both based on the examples provided in the documentation of the Unity ML-Agents toolkit.

In the first case study, the Vector Agent that used curriculum learning had consistently higher average cumulative reward for the duration of the training and in terms of the average entropy the Vector Agent that uses curriculum learning had the average entropy lower for most of the training except in the beginning where the Vector Agent that did not use curriculum learning had significantly lower average entropy. The reason for this behavior was not discovered in this thesis. This showed that using curriculum learning can result in improvements in the statistics of the training.

In the second case study, the Raycast Agent that used curriculum learning fared better only at the beginning of the training for both the average cumulative reward and the average entropy. At the end of the training both of the Raycast Agents had converged around the same values. The major difference between the Raycast Agents was that when curriculum learning was used, the training could have been stopped much earlier to achieve the same results as with the Raycast Agent that did not use curriculum learning. This showed that in this case, the usage of curriculum learning can result in quicker training.

For the evaluation of the first case study, the Vector Agent that did not use curriculum learning had better average completion time from all of the evaluations and had more successful evaluations while the Vector Agent that used curriculum learning had better average completion when only the successful evaluations were counted in the average. The Vector Agent that used curriculum learning had a larger percentage of its successful evaluations take less than 15 seconds which meant

that it had fewer mazes that it could complete but those it could, it did on average faster than the Vector Agent that did not use curriculum learning.

The Raycast Agents in the evaluation of the second case study had results that were as expected. The Raycast Agent that used curriculum learning was slightly faster in terms of the average completion time. The difference in average completion between the two Raycast Agents was slightly larger when only the successful evaluations were counted in the average and the success rate was also slightly higher for the Raycast Agent that used curriculum learning. This result was in line with the training results for both of the Raycast Agents.

As machine learning continues to be used in a larger variety of fields, it is only a matter of time until machine learning is widely adopted into video game development workflows. In the end, the only question is how it will be used. It might be as an alternate way to create the AI for a video game, as a way to automate content creation or in some other way. The video game industry might even decide that machine learning is not worth the effort: that the training takes too long or that the trained `Agent` does not act in the desired manner. But to even reach the point where these kinds of decisions are made, more studies and projects that create smaller, more prototype feeling games have to be made.

References

[1] A. Tervo. "Unity-maze-explorer-ml-agent". (2022), [Online]. Available: https://github.com/Ikaovret/Unity-maze-explorer-ML-Agent (visited on 05/27/2022).

[2] Unity Technologies. "Unity machine learning agents". (2020), [Online]. Available: https://unity.com/products/machine-learning-agents.

[3] ——, "Unity's terms of service". (2022), [Online]. Available: https://unity3d.com/legal/terms-of-service/software (visited on 03/23/2022).

[4] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects", *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[5] B. Liu, "Supervised learning", in *Web data mining*, Springer, 2011, pp. 63–132.

[6] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[7] I. C. Education. "Deep learning". (2020), [Online]. Available: https://www.ibm.com/cloud/learn/deep-learning (visited on 05/13/2022).

[8] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[9] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents", *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[10] A. Juliani, V.-P. Berges, E. Teng, *et al.*, "Unity: A general platform for intelligent agents", *arXiv preprint arXiv:1809.02627*, 2018.

[11] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning", *arXiv preprint arXiv:1708.05866*, 2017.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Human-level control through deep reinforcement learning", *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[13] OpenAI. "Proximal policy optimization". (2017), [Online]. Available: `https://openai.com/blog/openai-baselines-ppo/` (visited on 05/15/2022).

[14] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms", *arXiv preprint arXiv:1707.06347*, 2017.

[15] C. C.-Y. Hsu, C. Mendler-Dünner, and M. Hardt, "Revisiting design choices in proximal policy optimization", *arXiv preprint arXiv:2009.10897*, 2020.

[16] C. Berner, G. Brockman, B. Chan, *et al.*, "Dota 2 with large scale deep reinforcement learning", *arXiv preprint arXiv:1912.06680*, 2019.

[17] P. Soviany, R. T. Ionescu, P. Rota, and N. Sebe, "Curriculum learning: A survey", *arXiv preprint arXiv:2101.10382*, 2021.

[18] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning", in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 41–48.

[19] X. Wang, Y. Chen, and W. Zhu, "A survey on curriculum learning", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[20] S. Jin, A. RoyChowdhury, H. Jiang, *et al.*, "Unsupervised hard example mining from videos for improved object detection", in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 307–324.

[21] S. Narvekar, J. Sinapov, M. Leonetti, and P. Stone, "Source task creation for curriculum learning", in *Proceedings of the 2016 international conference on autonomous agents & multiagent systems*, 2016, pp. 566–574.

[22] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms", *arXiv preprint arXiv:1707.06347*, 2017.

[23] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor", in *International conference on machine learning*, PMLR, 2018, pp. 1861–1870.

[24] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation learning: A survey of learning methods", *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, pp. 1–35, 2017.

[25] J. Ho and S. Ermon, "Generative adversarial imitation learning", *Advances in neural information processing systems*, vol. 29, 2016.

[26] B. Baker, I. Kanitscheider, T. Markov, *et al.*, "Emergent tool use from multi-agent autocurricula", *arXiv preprint arXiv:1909.07528*, 2019.

[27] Unity Technologies. "Unity ml-agents python low level api". (2021), [Online]. Available: `https://github.com/Unity-Technologies/ml-agents/blob/release_18_docs/docs/Python-API.md` (visited on 05/09/2022).

[28] ——, "Training configuration file". (2021), [Online]. Available: `https://github.com/Unity-Technologies/ml-agents/blob/release_18_docs/docs/Training-Configuration-File.md` (visited on 05/08/2022).

[29] ——, "Using tensorboard to observe training". (2022), [Online]. Available: `https://github.com/Unity-Technologies/ml-agents/blob/release_18_docs/docs/Using-Tensorboard.md`.

[30] J. Hocking. “Procedural generation of mazes with unity”. (2018), [Online]. Available: https://www.raywenderlich.com/82-procedural-generation-of-mazes-with-unity%5C#toc-anchor-006 (visited on 03/23/2022).

[31] Í. Goulart, A. Paes, and E. Clua, “Learning how to play bomberman with deep reinforcement and imitation learning”, in *Joint International Conference on Entertainment Computing and Serious Games*, Springer, 2019, pp. 121–133.

[32] Thunderful Games. “Source of madness”. (2022), [Online]. Available: https://sourceofmadness.com/ (visited on 05/20/2022).

[33] A. Sestini, A. Kuhnle, and A. D. Bagdanov, “Deepcrawl: Deep reinforcement learning for turn-based strategy games”, *arXiv preprint arXiv:2012.01914*, 2020.