UNIVERSITY OF TARTU

Institute of Computer Science

Computer Science Curriculum

Tanel Marran

Developing a Volleyball Game with an AI Opponent Using Reinforcement Learning

Bachelor's thesis

Supervisor Tambet Matiisen

TARTU 2021

Developing a Volleyball Game with an AI Opponent Using Reinforcement Learning

Abstract: This thesis integrates reinforcement learning into a game development project by creating a competitive volleyball game, where the user can play against an artificial intelligence (AI) trained using reinforcement learning techniques. The work elaborates on what reinforcement learning is, brings forth some of the challenges of adding machine learning to a game, describes the development environment Unity and its machine learning package ML-Agents as well as analyzes the finished game and its AI.

Keywords: machine learning, reinforcement learning, game development, self-play

CERCS: P176 Artificial intelligence

Võrkpallimängu ning stiimulõppega loodud tehisintellekti arendamine

Lühikokkuvõte: Bakalaureusetöö eesmärk on luua võrkpallist inspireeritud arvutimäng ning lisada sellele mängule vastane, mis on loodud kasutades stiimulõpet. Töö annab ülevaate masinõppimisest ning selle erinevatest liikidest, kirjeldab võimalikke takistusi, mis tekivad, kui lisada mängule masinõppe elemente, annab ülevaate arenduskeskkonnast Unity ja sellega kaasnevast masinõppemoodulist ML-Agents ning kirjeldab ja analüüsib töö raames valmis saanud mängu ning selles oleva tehisintellekti pädevust.

Võtmesõnad: masinõpe, stiimulõpe, mänguarendus

CERCS: P176 Tehisintellekt

Table of contents

1.Introduction

Machine learning is a subset of the artificial intelligence field of computer science. It deals with methods and algorithms used to train an artificial intelligence model to perform some task without explicitly needing to program how that task is performed. These models can learn purely through observation of existing data or reinforcement of favorable patterns (Bishop, 2006). The field has garnered increasing attention over the past decade. This is for a good reason – the potential benefit of machine learning in multiple fields of study, from biology to data analysis, is great. For instance, over the past few years, machine learning has enabled researchers to tackle the problem of protein fold recognition in new ways, which has exciting implications for the future development of medicine (Callaway, 2020). The use of machine learning in this field has already yielded a breakthrough in fold recognition thanks to DeepMind's AlphaFold AI (Jumper, 2020).

However, machine learning has applications outside of scientific fields as well. Machine learning can also be used for entertainment and leisure, as is the case with video games, which can utilize machine learning in all manner of ways, from simplifying AI development to creating entirely unique experiences. For example, *Bardo Composer*[1] is a system that translates human speech to a dynamic soundtrack that matches the emotion of the players' conversations using machine learning algorithms. Bardo (Ferreira, Lelis and Whitehead, 2020) is intended to be used during tabletop role-playing games; however, the technology could be implemented in a video game to create a stronger sense of atmosphere while reading text inside the game. Another game from recent years that uses machine learning in an interesting way is *AI Dungeon 2*[2], created by Nick Walton. *AI Dungeon 2* is a game that uses a machine-learning AI to create a completely unique text-based adventure for the player. This is an example of a game that could not have been realized had it not been for machine learning; it is the essence of the entire game.

---

1 https://github.com/lucasnfe/bardo-composer

2 https://play.aidungeon.io/

This thesis also explores the application of machine learning in video games. The goal is to create a competitive, volleyball-inspired game and use Unity's machine learning agents (Juliani et al., 2018) to train a model that can play the game and compete against a human player.

The goals of this thesis are:

1. Design and develop a game that is playable on its own.
2. Make the game suitable for training a machine learning agent.
3. Train a machine learning model to be competent and competitive at the game. The model should be able to score points, defend against the opponent's attacks and ultimately win the game. The model's performance should be evaluated based on player experiences, feedback, as well as game recordings.
4. Analyze the process of development and identify the unique challenges of developing a game with machine learning in mind.

The thesis is split into three main sections, excluding the introduction. Section 2 (Background) gives an overview of reinforcement learning, self-play and other vital concepts used in developing the game of this thesis. Section 3 (Development) elaborates on the process of developing the game, giving insight into different iterations of the game, how the game changed and what challenges arose when implementing machine learning into the project. Section 4 (Results) will go over the results of the thesis. This section includes the analysis of a player survey, a description of the final released product as well as recorded statistics describing the model's performance against both new players and the developer. The Appendix contains graphs detailing the results of the player survey, the code used in calculating ELO values, hyperparameter sets used in determining model improvement rates and the license of this thesis.

2. Background

It is essential to understand some theory behind machine learning to better approach the problem of creating a competitive volleyball artificial intelligence. Knowing how reinforcement learning works and how it differs from other machine learning methods, choosing relevant observations for a model and creating a game suitable for the Unity ML-agents package – all these subjects and many others need to be discussed to achieve the goals of this thesis.

2.1 Supervised and unsupervised learning

Machine learning has grown and branched out in many ways. Random forest models, support vector machines, nearest-neighbor estimators, neural networks – the ways in which a computer can be taught to perform an action with little supervision are numerous.

When first introduced to the concepts behind machine learning, a good starting example is usually of a model that is trained on pre-existing data. For instance, the Seeds dataset[3] is a collection of 210 samples of different observed wheat kernels. Each observation records the same seven attributes of the kernel's internal structure as well as the known species of the sample. This data is then used to train a machine learning model that finds a correlation between the attributes and the species of the wheat seed. The trained model can then be used to predict the species of an unobserved sample, given the seven characteristics (Charytanowicz et al., 2010). While there is a lot more that goes into making a good classifier (data pre-processing, machine learning algorithm selection, combating overfitting), the general principle is the same – the machine learns to generalize a relationship by observing existing data. This generalization can then be used to classify unobserved data. Training a model in this fashion – showing it examples of attribute-label sets – is called supervised learning (Bishop, 2006).

There also exists a subset of machine learning that uses unlabeled examples to train a model. This, in contrast, is called unsupervised learning. Where the aim of supervised learning is to learn an existing pattern, the goal of unsupervised learning is to find hidden structures in data. The model is given unlabeled data and its task is to discover the underlying structure in data that can be used

[3] https://archive.ics.uci.edu/ml/datasets/seeds

in decision making. In essence, unsupervised learning attempts to clean data, either by reformatting it or removing unnecessary data, as is the case with dimensionality reduction (Ghahramani, 2004).

2.2 Reinforcement learning

The naming convention of supervised and unsupervised learning is a bit misleading – it suggests that all types of learning can fit under these two categories, which is not true (Sutton and Barto, 2018). The approach implemented by ML-Agents (Juliani et al., 2018) is called reinforcement learning and it has an entirely different approach to teaching a computer how to perform a task. It does not use labeled examples to fit a model, nor does it use unlabeled data akin to that of unsupervised learning. Instead, it works to increase a numeric reward function.

Reinforcement learning works based on three core principles. The first characteristic is that any action taken during training has a cascading effect on later actions. This is because reinforcement learning does not work off pre-existing data. Instead, the goal of reinforcement learning is to learn how to interact with an environment to maximize a reward function – a value that designates how well the model is currently performing (Sutton and Barto, 2018). Therefore, if the model learns the inputs needed to increase the reward function to some extent, then the model can use that knowledge to decide on a later input to increase the reward function further.

The second characteristic is that while training, the model is not given any guidance on what action to take. Instead, only the consequences of actions are acknowledged using the reward function (Sutton and Barto, 2018). When it comes to interactive environments and reinforcement learning, there is no 'correct' action the model could perform. Video games are open-ended environments where many different approaches to achieving a goal can be used. Who is to say that a more cautionary approach to a problem posed in a game is less advised than a more aggressive one? In reinforcement learning, only the outcomes of actions can be rewarded. These rewards then dictate how the model learns and develops (Sutton and Barto, 2018).

The third feature of reinforcement learning is that one instance of training spans an extended period (Sutton and Barto, 2018). While predicting the species of a wheat kernel from its features can be done in one step, it is not so with reinforcement learning. In a game, it may take some time until somebody scores a point, or a consequence of an action warrants an increase in the model's reward. The model needs time to experiment with different actions before starting over. This does have the

added consequence of training taking more time since the environment needs to be simulated; however, in the case of video games, the simulation of the environment can be sped up by increasing the rate at which time passes in the environment.

These three principles work together in the following way to train a model:

1. Once training has begun, the model will take steps in the environment. Each step is coupled with observations of the environment (i.e., vectors of key features such as the position of the player).
2. The received observations get passed through the model's internal neural network and are used in determining what action the model should take (the output layer). The model will repeat taking steps for the extent of training and any favorable outcome is rewarded by the reward function. Optionally, unfavorable outcomes can also be penalized by giving a negative reward signal on their occurrence.
3. Over the course of training, the model will attempt to learn what actions need to be taken for different situations (observations), that will lead to the largest reward.

If the reward function is effective enough, this will lead to the emergence of intelligent behavior.

Some interesting issues arise when you consider reinforcement learning as opposed to other learning paradigms. One of these issues is the exploration-exploitation dilemma. When using reinforcement learning, there are two things the model must do. It must explore the environment and its possibility space to find actions that have a favorable outcome (increase in the reward signal). However, the model must also exploit currently known actions and behaviors to increase the reward signal. The dilemma arises when you consider that the model cannot pursue either objective without failing the other. If the model exclusively focuses on exploration, it will lose out on valuable reward that it could have otherwise obtained had it exploited known behaviors. On the other hand, if the model favored exploitation, it would fail to explore the possibility space which might contain an even more lucrative, yet undiscovered behavior that could then be exploited. This dilemma is yet to be resolved (Sutton and Barto, 2018), however there exist certain measures that help alleviate the consequences of the explore-exploit dilemma.

As with any machine learning practice, reinforcement learning has numerous hyperparameters which can be tuned to better the model's performance. For example, the Unity ML-Agents package

– which is further discussed in Section 2.3 – allows the developer to tune the balance between intrinsic and extrinsic reward signal strengths. Extrinsic rewards signals are the reward signals set up by the developer and are specific to the model being trained and its environment. For example, an extrinsic reward signal in volleyball would be scoring a point or properly serving the ball over the net.

On the other hand, intrinsic reward, also known as curiosity, is a reward signal that is not directly coupled with the environment or the model being trained. Curiosity is given to the model for exploring more of its environment. It is inspired by how people can stay motivated to pursue a hobby or other activity purely because the activity is engaging and fulfilling. Curiosity as a reward signal is used most often in environments where extrinsic reward is given sparsely. Here, curiosity helps the model navigate more unknown behaviors by rewarding exploration and ultimately leading the model to new extrinsic reward (Aubret, 2019; Burda, 2018).

Curiosity is not only useful for environments with sparse rewards – it can also be used when designing a good extrinsic reward function proves difficult (Aubret, 2019; Burda, 2018). An example of a game that designing a reward function would be difficult for is Outer Wilds[4] by Mobius Digital. The game is a space exploration game set in a meticulously designed solar system, riddled with mysteries. The final objective of the game can be completed in less than 22 minutes; however, the bulk of the game involves gathering the knowledge needed to complete this final task. Designing a reward function for a game such as this is difficult since there really is only one sequence of actions that can be taken to complete the game, thus making the actions that would give any reward very specific and sparse. This reward function also assumes that the aim of the model is to complete the game. If the goal of the model were to learn how to navigate the solar system and its various planets, then designing a reward function for that goal is equally difficult. When designing a reward function is both difficult and the rewards too sparse, as is the case with the above example, curiosity is a great tool to help reward the agent.

There also exists the problem of credit assignment. The end goal of any competitive game is to be the victor at the end. How can it be determined what the steps required to win a game are? There is rarely a single action that determines who wins a game. More often, a game's result is determined

[4] https://www.mobiusdigitalgames.com/outer-wilds.html

by every step taken during play. For example, in the case of volleyball, it is not only the offensive power of a team that counts – a team's ability to defend their side of the court also plays a sizable role in winning the game. How should these individual actions be rewarded in the case of reinforcement learning? Does each action participate equally in determining the result of the game or are certain actions more important than others? These questions are at the core of the credit assignment problem (Minsky, 1961). Credit assignment is a vital thing to consider in reinforcement learning, since – much like with competitive games themselves – a well-designed reward function that suitably rewards the model over the course of training will be crucial in improving the model's performance.

In conclusion, reinforcement learning is teaching a model in an open-ended environment how to perform an action by rewarding wanted behavior and optionally punishing unwanted behavior. In this sense, it is not much different from the famous Skinner box experiment, where a subject is taught that performing a certain task rewards them with a treat, thus reinforcing that behavior (Skinner, 1938). The model receives important information about its environment by way of observations, that get past through an ever-improving neural network, which outputs the next action the model will take. This process gets repeated over an extended period of time to train the model.

2.3 Unity ML-Agents

Unity is a 3D development software used widely for making games and other interactive experiences[5]. It is a flexible platform with numerous different packages that add new functionality or other improvements to the editor. One such package is the Unity ML-Agents toolkit, which allows the developer to train a variety of different intelligent agents (Juliani et al., 2018). Since the ML-Agents package is the cornerstone of this thesis's artificial intelligence development, it is important to understand how the package works, what it can do and how it would work within the context of training an agent to play volleyball.

Since Unity is a general-purpose platform for developing interactive experiences, it is perfectly suited for machine learning. It has powerful graphics rendering and physics simulation capabilities,

5 https://unity.com/

making it useful for simulating real-world phenomena for the purposes of training a model or generating imagery that can then be used as a training set for an image recognition model. Unity also supports writing scripts using C#. This functionality can be used to write custom physics, game logic and interactions, which consequently makes Unity great for developing games that have a machine learning element to them, as is the case with the project in this thesis.

The core components of the ML-Agents package are:

1. Sensors. These collect information about the current environment and serve as inputs for the machine learning algorithm. Sensors can collect visual data, results of ray-casts, or other arbitrary information from the scene, such as a vector describing the position of an entity in the scene.
2. Agents. These are components which can be attached to GameObjects (instances in the Unity scene hierarchy) that the machine learning algorithm controls. Observations are taken in and in turn, the algorithm performs an action. In the case of a volleyball agent, this would mean moving the GameObject around, making it jump or hit the ball.
3. The Academy. This is a singular object that exists to keep track of the machine learning simulation and manage the agents. It can also alter the environment while the simulation is running, which is a useful tool for training. As an example, the Academy could alter the starting positions of players in the simulation to force the agent to learn how to deal with similar situations in a real match.

The other half of the ML-Agents implementation is the Python package. This provides the machine learning algorithms used to train the models in Unity as well as perform communication between the network and Unity.

In practice, the workflow of using ML-Agents is rather simple. First, a Python Virtual Environment must be set up containing all the required dependencies. Assuming a Unity project has already been made, the next step is to install ML-Agents into the project, after which it is simply a matter of adding the machine learning component to the appropriate GameObjects, passing in the

observations of the environment to the agent and defining a reward function. ML-Agents also has thorough documentation describing this entire process in far greater detail[6].

2.4 Self-play

One common problem when using machine learning for adversarial play is giving the agent a sufficient challenge. As the name implies, reinforcement learning is about reinforcing behaviors in AI similarly to how people learn. If the opponent is too good, then the agent will have a hard time understanding how it should play since most attempts would lead to failure. Conversely, if the agent can beat its opponent with minimal effort, it will have no reason to improve further or develop new, interesting strategies. Furthermore, how would one go about training a model to play an adversarial game? It cannot be expected for there to always be a human player opposing the AI for training due to the astronomical effort it would take for the human player to actively engage with the AI during the entire training process, which can take hours if not days. It is also not reasonable to create an AI to oppose the machine learning agent, since that would defeat the entire purpose of training a model to play the game in the first place.

The solution to these problems is self-play. Self-play is a reinforcement learning technique employed commonly in adversarial games that pits the training agent against its previous versions. After training the model for a while, a snapshot is saved of the model at that point. This snapshot can then be applied to the opposing player while training the model. Self-play is a great method for teaching a model how to play symmetrical, competitive games, since it ensures both a comfortable challenge as well as an elegant implementation for the opponent (Cohen, 2020). Since the model is being trained regardless, it is very convenient to use versions of the model as opponents for training. Furthermore, since we can choose which snapshots of the model to save into the pool of versions to use for training, we can also ensure a balanced training environment. If we use recent iterations of the model, both players will be of equal skill level, giving the agent the best environment for further improvement. Self-play can also combat overfitting by saving multiple snapshots of the agent. Certain versions of the model might have deployed different

[6] https://github.com/Unity-Technologies/ml-agents/blob/release_15_docs/docs/Readme.md

strategies. Therefore, if the model is trained against a variety of different playstyles, it helps the model from overfitting to combat one certain type of player (Cohen, 2020).

Self-play has also been shown to facilitate completely new behaviors in certain games. Many games have existed long enough for numerous strategies and techniques to emerge, only to be defeated by newer, even better strategies. These games have a history that is always taken into consideration when introducing the game to somebody new. Therefore, it is interesting to consider how a player would develop were they not already introduced to these well-established strategies. This question was explored by the model AlphaGo Zero, a machine learning agent taught to play the game *Go* while only ever being given the rules of the game. The model had no prior examples to learn from, so it was never influenced by the way humans play *Go*. This resulted in the model not only learning advanced strategies used by professional players but surpassing this knowledge by developing new techniques (Silver and Hassabis, 2017).

2.5 Result metrics

One of the goals set up in the introduction of this thesis was to create a volleyball machine learning agent that was competent and competitive. Determining whether a model fits these two characteristics is not a straight-forward task. The most intuitive answer, especially in game development and reinforcement learning, would be visual observation and testing. Since the end-user would play against the model, the best solution to see if the model works is to play against it. If the model can provide a sufficient challenge to their opponent by scoring points, defending their side of the court and being resilient to exploitative tactics by adjusting to their opponent's playstyle, then the model can be considered competent and competitive. However, this approach is most useful once a working model has already been developed. It is not a method that can be comfortably applied while the model is still learning. It also takes up more time since a real player needs to test the model by playing against it in real-time. Therefore, result metrics should be set in place that can be observed throughout training. These metrics would not replace testing the model by playing against it. Instead, they allow the developer to monitor how the model is doing without interrupting the training or testing it by hand. If a metric is showing signs that the model is not doing very well, the developer can restart the training with new parameters, therefore saving time by not letting a poor model train for an extended period with no results to show for it.

One of these metrics and possibly the most flexible one is cumulative reward. Cumulative reward shows how much reward the model was given during one attempt (also referred to as an episode in the ML-agents package) (Juliani et al., 2018). For example, training a model to play tic-tac-toe might involve a training environment where three matches were played in succession. If the model wins one of the games, it gets a reward signal of 1 unit. If a match ends in a tie, the reward signal is 0 and if the model loses, they get a reward signal of -1, which is effectively a penalty. During training, cumulative reward can be observed and the model's performance can be deduced from it. If the cumulative reward is 3, that means the model wins every game of the three in one episode and therefore is performing well. If the cumulative reward is -3, the opposite is true.

Observing this metric is relatively intuitive and can be useful in many reinforcement learning problems. However, it is essential to take into consideration both domain knowledge as well as the design of the reward function when interpreting these results. Since tic-tac-toe is a game with a finite action space (games are short and the total game space is a 3x3 grid), it is possible to play the game in a way that ensures a player never loses (Crowley, 1993). Therefore, if two perfect models were to play against each other, the cumulative reward would be 0, which might suggest the model is not doing well but is not true. Observing cumulative reward might also be misleading if the reward function is not designed well enough, leading the model to exploit a strategy that gives them a lot of reward but does not really align with the objectives of the game. For example, consider an arbitrary game where the model is rewarded for scoring points and for defending against attacks. The aim might be to score points and ultimately win the game. However, if the reward signals are not balanced well enough, the model might develop a strategy of allowing the opponent to relentlessly attack while itself defends each attack, therefore gaining a lot of reward. Cumulative reward as a metric also fails in specific self-play scenarios. Since self-play involves the model playing against recent iterations, the players will be equal in skill most of the time, leading to many games ending in a tie. If the reward function is designed similarly to the above example of tic-tac-toe, it will lead to the same issue as before. The model might be improving, but it is not reflected in an increase in cumulative reward.

The ELO rating system is a good metric to observe in a self-play environment. It does not have the same pitfalls as cumulative reward since it considers each player's relative skill level. The ELO system is commonly used in ranking chess players or sports teams and it can also be used to predict

the outcomes of matches. The ML-Agents package has ELO rating included and as the model improves during self-play, the ELO rating also increases (Juliani et al., 2018). There are many methods for calculating ELO rating and the following will outline one of these methods:

1. Given two arbitrary players' ELO scores R_A and R_B, calculate each players' expected result. This value ranges from 0 to 1, with 0 meaning defeat and 1 meaning victory.
   The expected result for player A would be:
   $$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}$$
2. Once the game between players A and B has been concluded, compare the expected results to the actual result (which is either 0 or 1 for a loss or win respectively). Adjust each player's ELO rating with the following formula:
   $$R'_A = R_A + K(S_A - E_A)$$
   Here, S_A is the actual result of player A. K is a constant that determines the rate at which ELO changes per game played. This value is adjusted to fit the game best. A common value used in chess is K = 32 (Aversa, 2019).

One issue with ELO rating is that it is not necessarily easy to interpret. Cumulative reward is rather easy to interpret – if the reward function rewards 1 unit for each successful action, then a cumulative reward of 100 units is a great result. However, in the case of ELO, the rating on its own does not give any substantial information about the performance of the model since it is not referencing anything – it is a relative metric. Given some arbitrary ELO rating, that rating is only useful after the respective model has been observed and its performance evaluated by a real person. In the case of this thesis' game, an increase in ELO was good simply for identifying that the model is improving, not necessarily for identifying whether the model was good or not. Still, there was one other metric that proved even more useful in this thesis than the ELO rating.

The length of each episode is a useful metric for determining the performance of a model, but it requires proper domain knowledge before it can be interpreted. Using the game of this thesis, an increase in episode length signifies an improvement in the model, since that means the players can keep the ball in play for longer. However, examining real life professional volleyball, each individual point is determined rather quickly – it does not take noticeably longer for a point to be earn at higher levels of play, as opposed to beginner or intermediate levels. Knowing this, it is

important to consider if this also applies to our volleyball game. Perhaps as the model improves, the total length of each episode will still stay the same. An increase in episode length might not also mean the model is improving. Consider a model that is learning how to push a box to some specific location. At the start of training, the model will struggle with completing this task and the episodes will be long. Conversely, as the model trains and improves, they will complete their task faster and faster.

3.Development

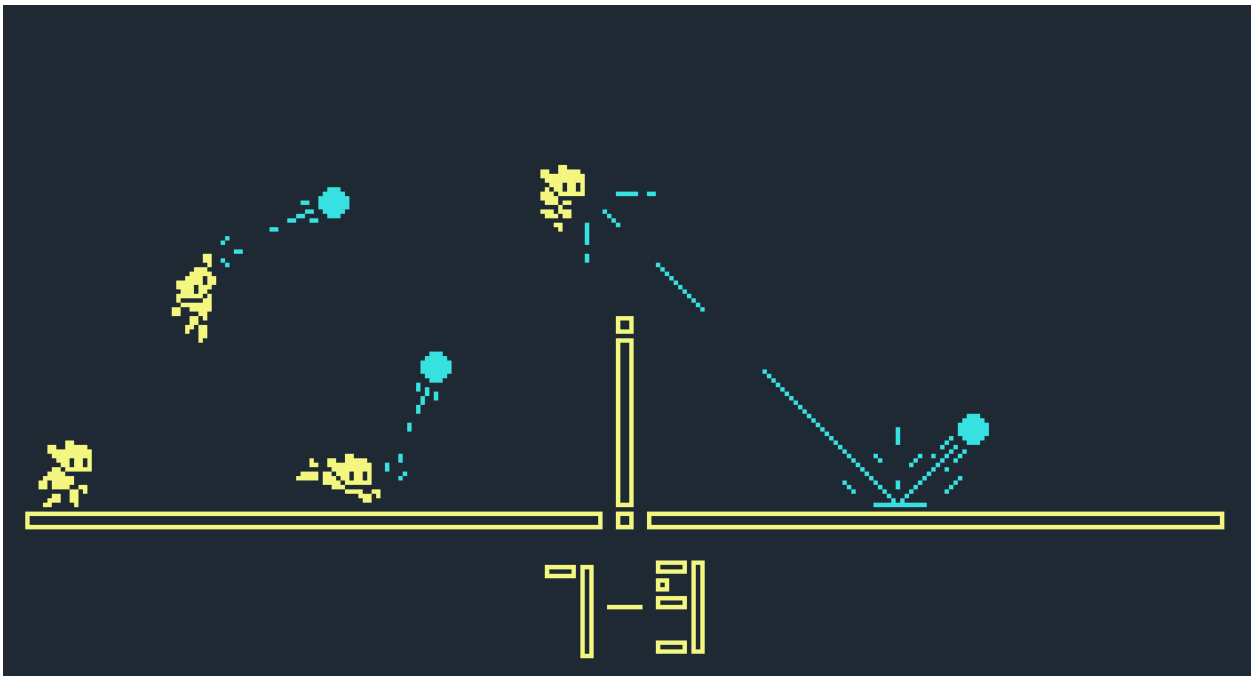


Figure 1: Initial concept art for the game

An important step that comes before training an artificial intelligence to play a competitive volleyball game is creating that game itself. The concept for the game existed before considering it as a thesis work involving machine learning. The initial concept art for the game can be seen on Figure 1. It is interesting to view this concept art in retrospect, since the design of the game changed so much from both a visual and mechanical perspective over the course of iteration.

3.1 Design

While concept art is a great jumping off point for development, before the game can be made, its design must also be considered. Considering design questions allows a game to be tailored to different player experiences, such as encouraging creative expression and thinking (Hall, Stickler, Herodotou and Iacovides, 2020). For this project, the most important aspects of the design were the following: what is the goal of the game, what mechanics need to be implemented, how the player interacts with the game world and how to ensure that the environment is suitable for machine learning.

Out of the four questions posed above, "What is the goal of the game?" is probably the simplest to answer. The setup of the game is as follows:

1. The game environment is a large rectangular room with a small wall in the middle of it that acts as the net in volleyball.
2. Two player characters are placed on either side of the net an equal distance away. They can only move around on their side of the net.
3. A ball is set in play that both players can interact with.

The goal of the game is to interact with the ball in such a way that it touches the ground on the opponent's side of the court, granting the player a point. A consequence of this goal is also having to protect your side of the court from the opponent's attempts to score a point.

The most fundamental mechanics that would need to be implemented are movement for both the ball as well as the players and the interaction between the ball, the players and the environment. The interaction with the environment is simple – the players and the ball should not be able to fall outside of the playing environment and the ball should be able to detect when it has collided with either side of the court in order to tally up points.

The interaction between the ball and the player is a bit harder to define since there are multiple different approaches that could be implemented. The player could have to physically touch the ball in order to interact with it. This has the benefit of not needing any extra inputs to work; however, making it work in a way that is predictable to the player may become a challenge. If the ball interaction is too granular and requires substantial precision on the player's part, then it may feel hard to use and unpredictable. However, if the interaction is too restricted, then the player may not have enough fine control to manipulate the ball in the way that best fits their playstyle. These same points also apply to training artificial intelligence – the more complex and granular a problem is, the harder it is to fit a sufficient model for it. As the difficulty of a problem increases, either more data or a more complex model would be needed (Xhu, Vondrick, Fowlkes and Ramanan, 2016). Another problem arises when you consider that the ball could get stuck between the player and the walls of the environment, given that the player physically interacts with the ball. These collision edge cases, while not very common, are still worrisome since they affect the consistency and predictability of the game for both human players as well as machine learning models.

An alternative approach would be to use an input that allows the player to hit the ball. If the ball is within a certain distance of the player when the input is triggered, the ball will fly out in the direction away from the player. This would have the benefit of removing the edge cases with collision mentioned previously and be easier to implement in a predictable way, though it comes with a large drawback. With the physics-based approach, a training machine learning agent would only need to learn to position themselves relative to the ball in order to interact with it. However, with this input-based interaction, in addition to positioning, the model would also need to learn the timing for a successful hit, which adds an extra layer of complexity to training. While there are more ways of design these interactions, these are the two that were considered for this specific project – both were tested out through the process of iteration, the results of which will be discussed in a later part.

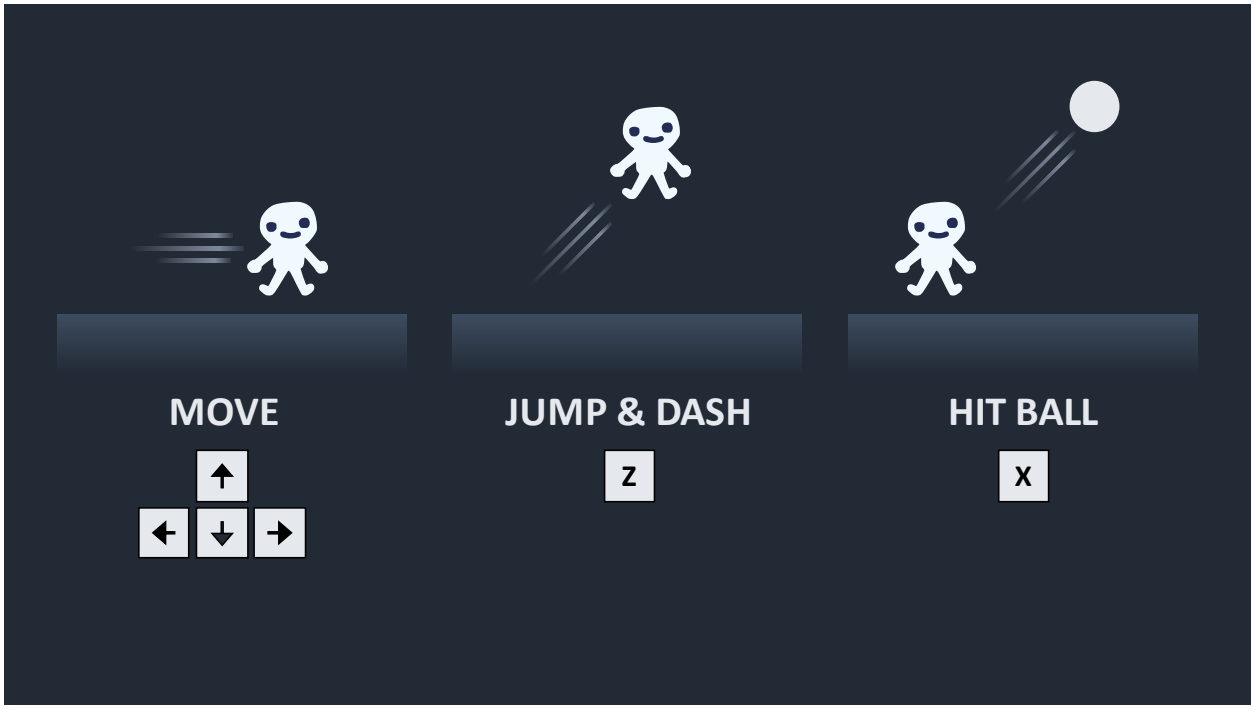


Figure 2: Player action illustration

With the core mechanics defined, the action space of the player also needs to be defined. In this thesis, action space refers to possible actions the player can perform by pressing an input key, respecting context sensitivity. For this game, the action space for the player is illustrated in Figure 2.

The following is a more in-depth description of each action:

1. The player can move left, right, or stand still.
2. The player can jump. Holding the jump button will result in a higher jump and lightly tapping the jump button will result in a lower jump.
3. The player can hit the ball. As mentioned previously, this was realized through physical interaction as well as a button input.
4. While in the air, pressing the jump key will result in a dash in one of 8 directions. The direction of the dash is determined by the combination of movement keys being held down (left, right, up and down). The player can only dash a certain number of times before needing to touch the ground to recharge the ability. This action gives the player more options in both offense and defense.

An extra consideration that is unique to a game using machine learning for artificial intelligence development is the model training environment. It is reasonable to assume that the environment where the end-user plays the game is the same as where the game's AI opponent is trained, but this is not strictly true. While there are examples of machines learning how to play an already released game, such as teaching a reinforcement learning model to play *Super Mario Bros.* (Heinz, 2019), creating a custom environment for model training using ML-agents can increase the speed of training as well as add more opportunities for supervision. To increase the speed of training, ML-Agents support running multiple simulations of the game in parallel, where each simulation contributes its own training experiences (Juliani et al., 2018). The developer can also set in place functions, mechanics, events and triggers that are specifically used for training. In this project, checking for when the ball crosses the net, when a player successfully interacts with the ball and randomizing player starting positions every round is mostly only useful for training and would not be needed in the end-user experience.

It is also important to note that the design decisions discussed above were not final and changed over the course of development and iteration. Furthermore, some of the above points were taken into consideration at the very start of development, while others were noted only over the course of iteration later into development.

3.2 Creating the game

The engine used to make the game in this thesis was Unity. It allowed the use of the Unity ML-Agents package, which helped accelerate the development of both the game and the machine learning artificial intelligence.

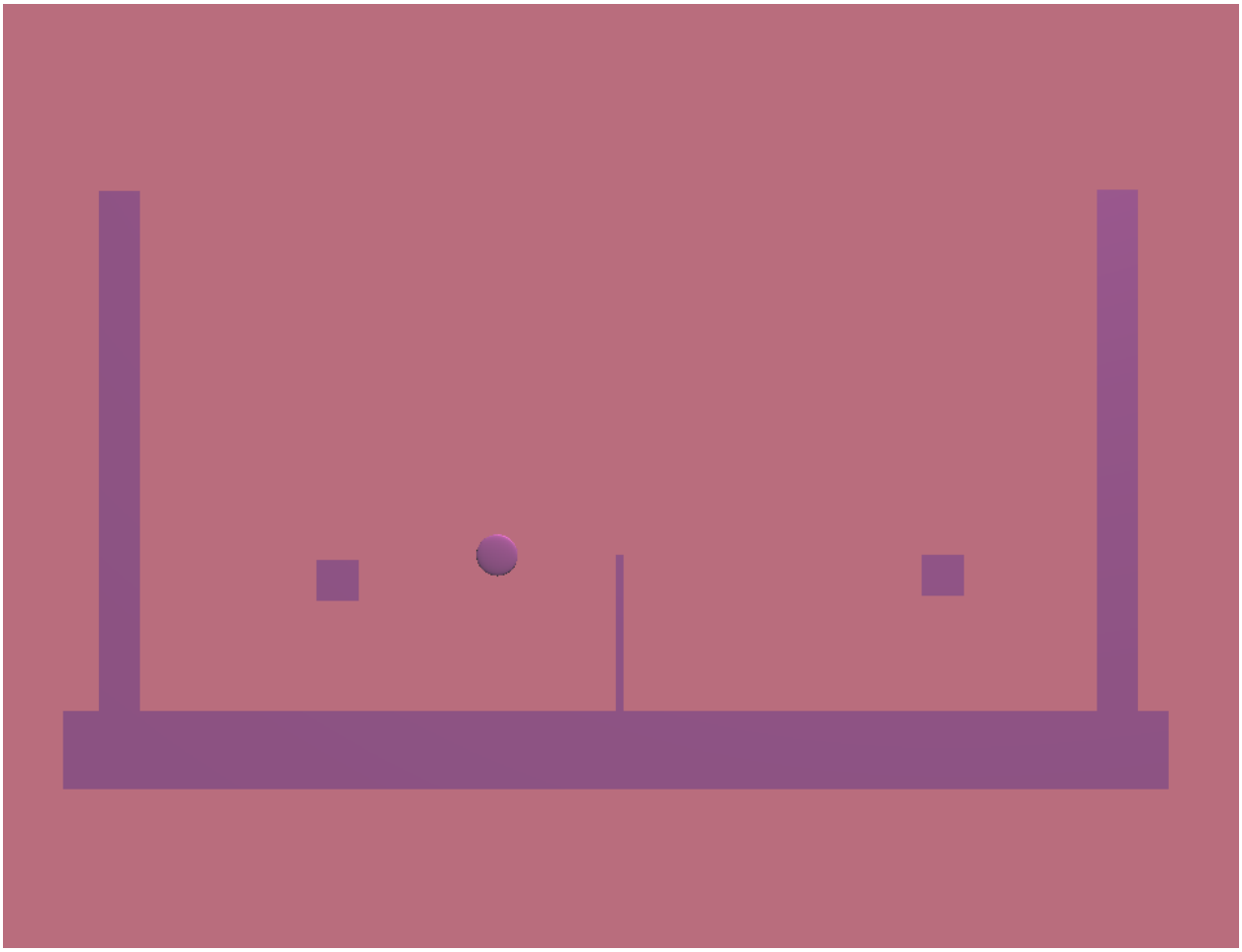


Figure 3: First iteration of the game.

The first playable version of the game is visible in Figure 3. The scene consisted of a game object aptly named "Game", under which most other objects were nested (the players, the ball and invisible detectors used to identify when the ball has touched the court). The walls of the environment were the only exception – they were not nested under the "Game" object. A visualization of the Unity scene's hierarchy can be seen in Figure 4. The first implementation used a physics-based approach to ball interaction discussed in the above section. The players were able to move, jump, dash and hit the ball by moving into it. The ball's trajectory would be determined by the contact direction and speed. Every round started with the ball falling on one side of the court, at which point the appropriate player would need to hit the ball over the "net" (the small rectangle at the center of the scene) onto the other side of the court.

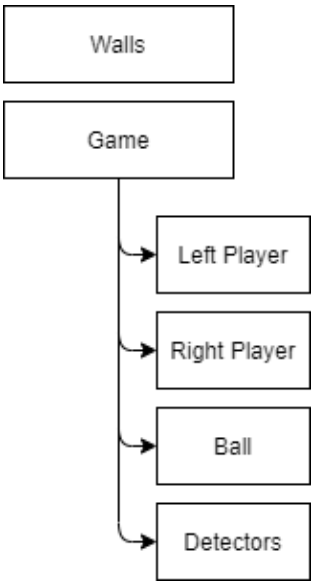


Figure 4: Object hierarchy.

The goal of this first version was to get a working demo ready as quickly as possible so that the first agents could start training. This, however, came at the cost of less refined environment design and game logic.

The major drawbacks of the first implementation were the following:

1. Inconsistent ball interaction. Due to the ball physically interacting with the player, it was possible to get the ball stuck in between one of the walls and the player. The player could simply move away from the wall to dislodge the ball, but it was still difficult to penalize this type of behavior, design around it or make it work in an interesting way for the benefit of the player.
2. Imprecise environment dimensions. The size of each players' court was different, making it easier for one player to score points than another, even if only marginal. It also made model training inconsistent since the artificial intelligence should be able to play on either side of the court without needing to train for it specifically.
3. Poor game logic generalization. The walls of the court were not parented under the general game object, making it difficult to move the game around in the scene. Furthermore, the logic of the game's state (how many points each player has, how the points are accounted

for and other events) was not grouped under one specific manager. Instead, it was sparsely divided among all actors of the game (the players, the ball, etc.), which made keeping track of the game's state and updating the implementation more difficult.

4. Inferior parallel play implementation. Instead of duplicating the game object and offsetting their position in the scene, the games running in parallel were stacked on top of each other and interactions between different games were prohibited. This not only made it difficult to observe the training in progress, but it also had the added uncertainty of never being completely sure whether the interactions between two different games really were being prohibited.

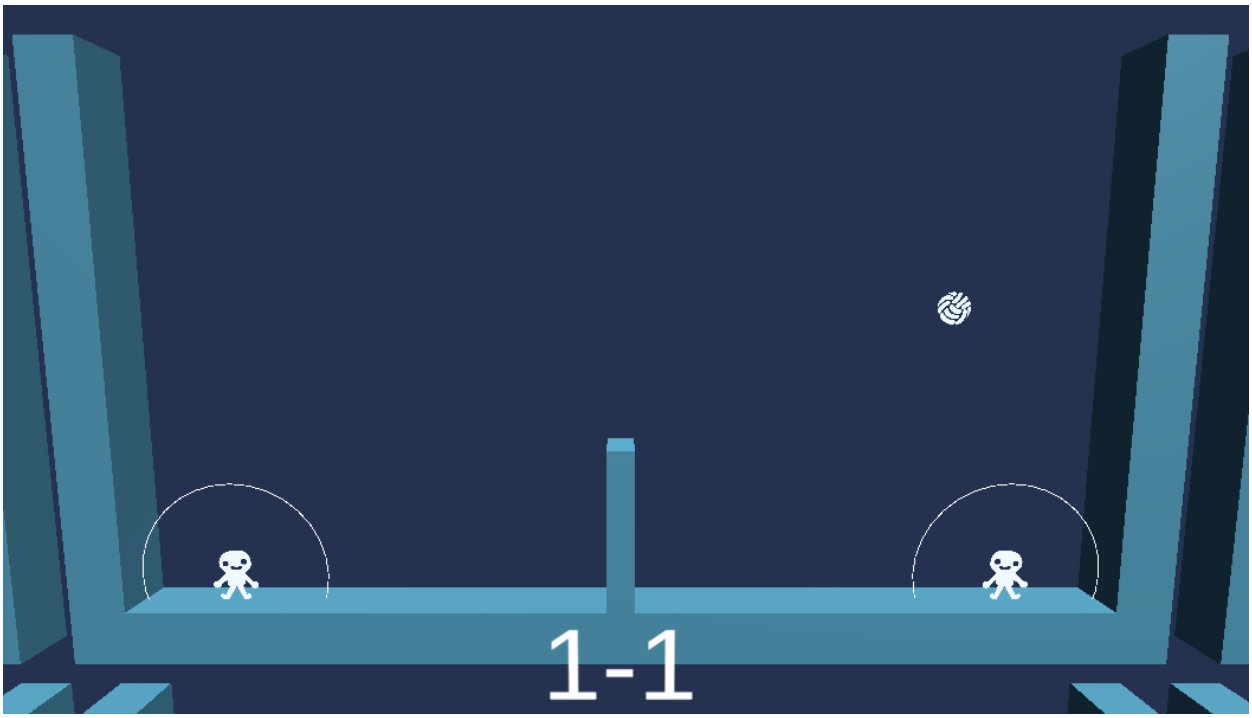


Figure 5: Second iteration of the game.

All the shortcomings were addressed in the following iteration of the game, which can be seen in Figure 5. The most notable difference is the environment itself. The game got many visual improvements, the most important of which was the counter at the bottom, signifying how many points each player has gotten. Both the players and the ball have their own sprites, which makes identifying what is going on far easier than it was before with the abstract, same-colored shapes. Alongside the visual improvements to the environment, walls and other hidden colliders are now

a part of the "Game" object hierarchy, making it easier to move things around or duplicating the game for parallel training.

The way the ball interacts with the player was also changed. Before, it was purely based on the direction and the power at which the player collided with the ball. However, in this iteration, the ball cannot physically interact with the player. Instead, it all works based on player input. The circles around each player signify the distance that they can interact with the ball. If the ball is inside the radius and the user/agent triggers the appropriate input, the ball will get a force applied to it pointing away from the player. The amount of force applied increases as the ball is closer to the center of the radius, making it possible to lightly tap the ball, send it flying with great speed, or anything in between. The circle changes its color depending on the current state of interaction:

1. The circle is white if the player has not triggered a hit interaction.
2. The circle is green if the player triggered the hit interaction. If the ball is in the radius during this state, it will be propelled away from the player. This state is maintained for a few seconds or until the ball is in the radius. This way, the player does not need to be too precise with their timing – they can press the interact button a few frames too early still have a chance to hit the ball lightly.
3. The circle is red when the hit interaction is on cooldown. During this period, the user cannot initiate a new hit until the cooldown has expired and the circle becomes white again. This state is initiated after the green state, regardless of if the hit was successful or not. This state was set in place to force the player to still need to time their hits to a degree.

In the first iteration, scoring points worked using two invisible collision areas near the ground on either side of the court. If the ball detected a collision with a solid object within one of those areas, the game assumed it had hit the ground and assigned one of the players a point. This, however, worked inconsistently and was simplified in the second iteration. Since the Game object hierarchy was set in place, the ball could check its relative position to the origin of the Game object, which was located at the center of the court at the base of the net. Therefore, points could be accounted for when the ball detected a collision below itself and its relative Y position was smaller than some threshold. If it was above the said threshold, then the ball collided with the top of the net instead and no points would need to be granted to either player.

The second iteration also has some more utilities that are useful when training the model. For instance, if the ball crosses over the net onto either side, an event is triggered with a Boolean value determining whether it crossed into the left side of the court or the right side. This can be used to reward an agent for serving the ball over the net. Utility functions for when an agent successfully hits the ball and for when they attempted to hit the ball but missed were also implemented in the second iteration. These triggers found similar uses as the event for serving the ball over the net.

The process of developing the game and its artificial intelligence were co-dependent. The design of the game and its mechanics changed with each attempt at training a model to play the game. For example, initially, the player character's movement input was continuous instead of discrete. This means that instead of the player either moving left, right or standing still, they could move at a fraction of their movement speed (this was only possible when a gamepad was used). This, however, meant that when training an agent, it also had to have a continuous action space, which increased both the training time and complexity. The change from continuous movement to a discrete solution was done rather early in development since it had the consequence of decreasing training time on any future attempt.

A similar change occurred with the dash action. Originally, dashing had its own button mapped on the controller, meaning the player could always either jump or dash and what the jump button did was not context-sensitive (whether the player was standing on the ground or not). While playtesting this feature, it became apparent that controlling the player character became increasingly difficult, since the number of buttons the player had to account for was too much for the game's fast pace. On numerous occasions, buttons for hitting the ball, jumping, or dashing got mixed up or dashing was completely forgotten in favor of a far simpler player experience. This issue was remedied with the dash action and jump action being merged under the same button input. Since the jump button did nothing while the player was already in the air, it was free to have a second, context-sensitive action mapped to it. There was a slight worry that due to the dash now being only available while in the air (since being grounded resulted in a regular jump), that the player would not be able to dash on the ground anymore. However, this worry was quickly resolved, since the player could simply double-press the jump/dash button – the first press would trigger a jump, lifting the player slightly off the ground, and the second jump would initiate a dash relatively close to the ground.

Name	Self Position	Opponent Position	Self Speed	Opponent Speed	Self Dashes Left	Opponent Dashes Left	Self Hits Left	Ball Position	Ball Speed
Number of Values	2	2	2	2	1	1	1	2	2
Type	Float	Float	Float	Float	Integer	Integer	Integer	Float	Float
Range	[-1, 1]	[-1, 1]	[-1, 1]	[-1, 1]	[0, 1]	[0, 1]	[0, 3]	[-1, 1]	[-1, 1]

Table 1: Model final observations

One aspect that did not change much over the course of iteration was the observations the model would receive of its environment. From the start, the set of required vectors was rather apparent – the model would need to know its own position and movement speed, along with the positions and movement speeds of its opponent and the ball. The only way these values changed over the course of development was normalization, which helped the model better handle the inputs. They also need to be perfectly symmetrical in relation to the center of the court to ensure that regardless of which side the AI plays on, it acts the exact same way. There were only ever three other observations that were considered – "player state", "available dash count" and "hits left" – the last two of which are used in the final version of the game along with the observations mentioned before. "Player state" was a value that signified different states of the player (jumping, moving, hitting, dashing etc.). This did not seem to affect the model's performance in any meaningful way, so it was ultimately removed. "Available dash count" expressed how many dashes the player was able to perform before needing to recharge the ability by touching the ground. This observation was important because otherwise, the model would have no way of knowing that they can only dash a certain number of times before needing to touch the ground. "Self hits" let the agent know how many times they could touch the ball before they needed to hit it over the net. If this value were not exposed to the model, they would constantly lose points due to not realizing that they can only interact with the ball a set number of times before needing to pass it over. The full list of final observations can be seen in Table 1.

While not strictly related to the design of the game, a helpful addition to the training environment that improved debugging speed was the addition of reward notices. Any time a model gets a reward signal, a visual indicator displaying the amount of reward gained or lost appears within the game. Figure 6 showcases this – a player gained 1 unit of reward for scoring a point. These indicators were a useful addition because they helped make sure that the machine learning implementation was working as intended and that each player was being rewarded for the expected outcomes. The implementation of these notices lead to the discovery and subsequent fix of several bugs that severely hampered the training of the models. After they were fixed, the first models that exhibited intelligent behavior emerged.

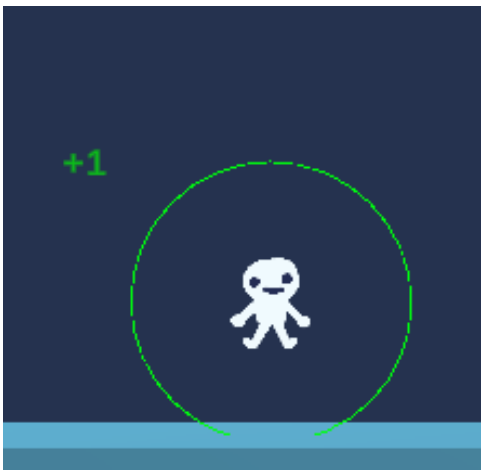


Figure 6: Player gaining reward and a notice being displayed.

At the end of the development cycle, it was also important to accomplish the first goal of the thesis, which was to make the game playable on its own. This meant that once the core gameplay was at a satisfactory point with no major bugs, shortcomings, or conflicts with machine learning and at least one model proficient at the game was trained, a rudimentary main menu was added with which a user could set up a game or view a short tutorial teaching the controls and mechanics of the game. The menu also allows the user to define who controls each player (human vs. human, machine vs. machine or human vs. machine) and how many points are needed for a win. This step was important because ultimately, the game should be enjoyable by an end-user, whether it be alone or with a friend.

3.3 Handling inputs

An important distinction that separates this machine learning project from the examples provided by Unity is that this game is also meant to be played by an end-user. Before any of the machine learning aspects were introduced, the game had to be created with the intent of making it playable by two human players. While good foresight was necessary to make sure no design decision was made, which would later complicate machine learning, it was still important to design the game such that it supported machine learning but was not directed by its inclusion.

One of these challenges on a technical level was handling player input. Generally, user input is already a difficult task to solve, since it defines how the player interacts with the game. Different controller options, key rebinding, accessibility concerns, number of inputs – all of these are issues that are universal to almost every game. When introducing machine learning to an adversarial game's development, the way inputs are handled must go through a level of abstraction to ensure that both a human player and the machine can interface with the game in the exact same way. This is also important if the game were to change in a dramatic way, whether it be that a new mechanic was introduced or the player's action space was tweaked. If this is not considered, the game's development time may increase, since any change to the core game also means the way the machine learning agent interacts with the game needs to be reworked.

The way this issue was tackled in this thesis was with the introduction of an input wrapper called the "Player Input Transformer" in the game's codebase. This transformer was able to take inputs from both the Unity Input System as well as the machine learning network's output layer and transform them into one, uniform interface which was then used to detect what actions a player character is taking. The transformer had methods that represented each input the player could perform – moving left or right, jumping, dashing and hitting the ball. It was also able to differentiate between an input starting and an input being continuously active. This was important when handling the player's jump, since a short press of the jump key resulted in a lower hop, while holding the jump key down for longer resulted in a far higher leap.

The benefit of this transformer was twofold. Firstly, implementing this transformer ensured that both a human player as well as a machine learning agent were on equal footing and that neither was able to perform something that the other could not, since regardless of whether it is a human or the machine learning agent, both of their actions would be transformed and handled in the exact same way. Secondly, this abstraction meant that if the game were to change in some way – either the speed of the dash was altered, a new double-jump mechanic was added or a change in the way that hitting works was introduced – no extra steps were needed to make these changes work with the machine learning agent. Of course, a new model would need to be trained to account for these changes or the size of the agent's output layer would need to be changed if new buttons or actions were introduced, but the implementation from a technical level would still be the same. If this transformer were not added, the machine learning agent would have interfaced with the game

directly, which could have led to inconsistencies in what a human player and the machine can perform.

Once this transformer was implemented, the way that the model's neural network handled its output layer was set in stone. The output layer was a vector of Boolean values corresponding to each button that a player would use to play the game. If a Boolean value was true, it correlated to that button being held down and vice versa for when that same value was false.

4.Results

This section will elaborate on the results of this thesis, whether the goals of the thesis were met, discuss the performance of the machine learning agent from both a play-driven as well as data-driven perspective and showcase how the models' training changed with different hyperparameters and reward functions.

4.1 The game and mechanics

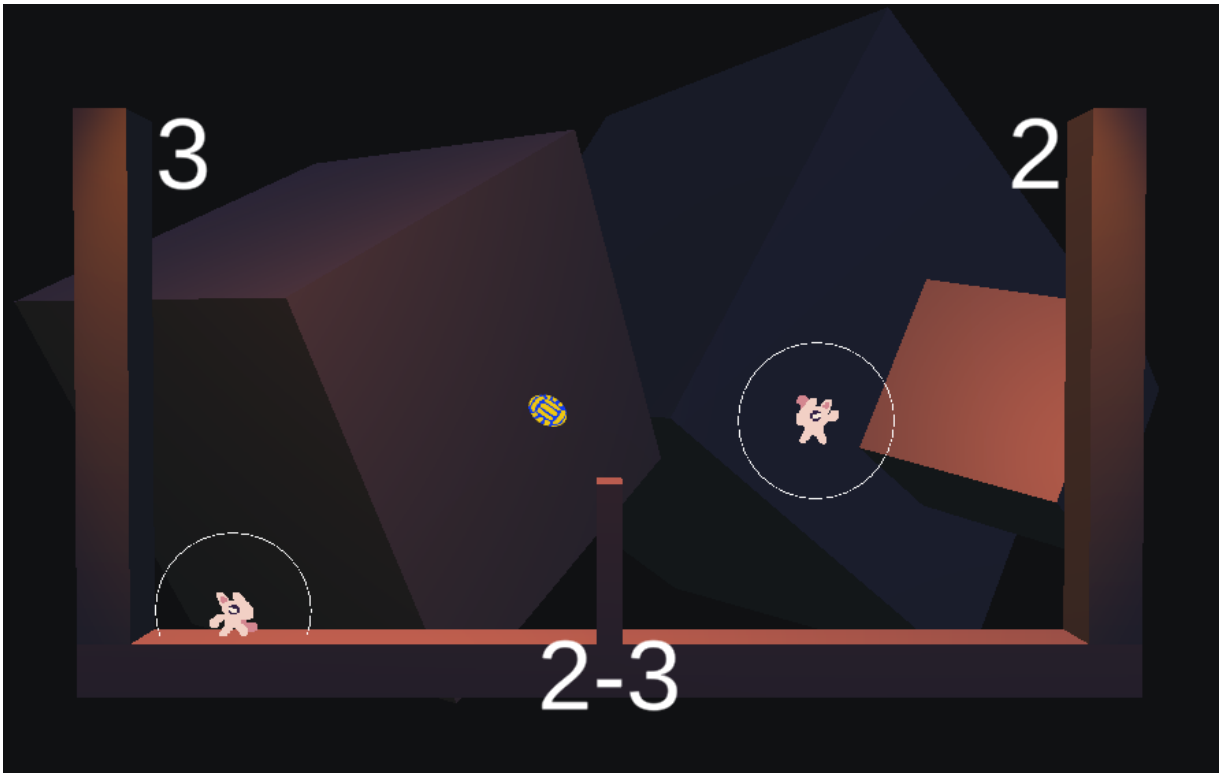


Figure 7: The final iteration

The game, named VOLL-AI, was published on the webpage itch.io[7] on the 3rd of April 2021. The full code of the game is available in a repository on GitHub[8]. Overall, the game came together well in the end, in both mechanics as well as presentation.

[7] https://forestherd.itch.io/voll-ai
[8] https://github.com/TanelMarran/Voll-AI

Figure 7 showcases how the presentation of the game evolved to fulfill the goal of the game being enjoyable on its own. The game has a stronger visual identity, with more involved lighting and hovering cubes in the background as a low time cost method of adding more visual interest to the scene. The game also has a few particle effects, an indicator to help the player understand where the ball will fly once hit and a couple of sound effects to make the whole experience more enjoyable. Furthermore, the game could be enjoyed in numerous different ways: either by playing against a friend, an AI or even setting both players to be controlled by the AI and watching a game unfold with no further input.

The mechanics of the game work rather consistently. The trajectory of the ball is easy to estimate and it bounces around in a predictable fashion. The premise of the game is simple to grasp and the short text-based tutorial is enough to make the mechanics of the game clear. The way the player interacts with the ball does not seem inherently flawed, but one consequence of hit strength increasing as the ball draws closer to the player character is that aiming becomes more difficult. This is because the ball flies away from the player at the center of the hit radius, and the angle from the center changes very quickly the closer the ball is to the player. On the one hand, this meant that the player always sacrificed strength for accuracy or vice versa, which was an interesting risk-reward relationship. On the other hand, this tradeoff might have just felt like an inconsistent frustration for some players.

When it comes to frustration, it also does not help that the controls of the game are quite tricky and hard to get accustomed to. The game's survey (which is further analyzed in Section 4.2), as well as feedback on the game's download page, brought to light that the controls were rather difficult to grasp. One player said that due to the game's quick pace, the number of buttons and actions to consider was rather overwhelming. Most players who responded to the survey said that while more time with the controls would improve their performance, they would never feel fully in control of the character.

The game also had two different difficulty settings for the AI, where the harder version of the AI was identical to the easier one in both hyperparameters and the reward function used, only having trained for a longer period. The analysis of whether a longer training time resulted in a more proficient model is discussed in Section 4.3.

One interesting consequence that emerged from the game's mechanics as well as its balance was the effectiveness of blocking. In volleyball, blocking is when an opposing player jumps in front of the ball right under the net in hopes of catching it before it comes over to their side of the court. This strategy can also be used in VOLL-AI with great results – so great that it even affected the viability of spiking the ball at all, because once blocked, it meant a virtually guaranteed point for the opponent. Blocking was also the one strategy that the AI learned rather quickly, regardless of hyperparameters. Some possible tweaks to weaken the strength of blocking could be allowing the player to always dash down towards the ground, giving them a chance to save the ball before it touches the ground. An alternative fix would be to reduce the time a hit is in its active state, making it easier to time a spike past the block.

4.2 AI analysis

Upon the game's initial release on the platform itch.io, it was coupled with a short survey accessible through the game's main menu that focused on players' experiences with the AI. The survey[9] consisted of 11 questions along with a 12th field, which allowed participants to upload game records which helped determine the AI's win rate. A total of 10 players gave feedback. The questions could be broadly grouped into having three main objectives – determining the AI's intelligence, strategies and how fun it is to play against. Some questions were added to the survey after the initial release of the game, so not every question has an equal number of responses. This section analyses the most important responses from the survey. The complete list of questions and answers can be found in Appendix I.

Before analyzing the results of the survey, it is important to note that the game's controls had an obvious impact on the experiences of the surveyed. The response from players was that the controls for the game were rather difficult, and with the game lacking a formal tutorial level or tempered difficulty curve, it is reasonable to assume that even winning a game against an inept machine learning agent may prove difficult. Since most surveyed only played the game for about four matches, it is important to consider whether the model truly is proficient at the game or were the players playing against it too unfamiliar with the mechanics themselves. To combat this, the results

[9] https://docs.google.com/forms/d/19m3D4TBl0mWHx9KZgPagnU03HLIZfzfYCsd-kqiGy-0/edit#responses

of the developer's games against the AI have been separated from that of the surveyed, since the developer has had the most experience playing against the AI and learning the controls.

	Games Won	**Average Lead**	**Win-rate**
Player	0	-	0%
AI	42	4.25	100%

Table 2: AI vs. Player results

The reward function used in the published game rewarded 1 unit of reward for a point and -1 unit of reward for a point loss, since during testing, this function resulted in the most adequate models. The results of the survey also support this claim, with 100% of all answers to the question of "How intelligent was the AI?" being rated either a 3 or 4 on a 4-point scale, with 4 meaning "very intelligent" and 1 meaning "not intelligent at all." Also, when asked, "Recall your games against the AI. In general, who won and by how much?", almost everybody recollected that the AI won most of the time and with a big lead, save for one player, who remembered the AI winning but not with a notable lead.

This is also supported by the recorded game results that each answerer submitted, visible in Table 2. Out of the 42 games recorded by 10 of the surveyed players, the AI won 42 games with an average lead of about 4.25 points. This amounts to a win rate of 100%. This also means that regardless of whether the players played against the easy or hard AI variant, the AI won regardless.

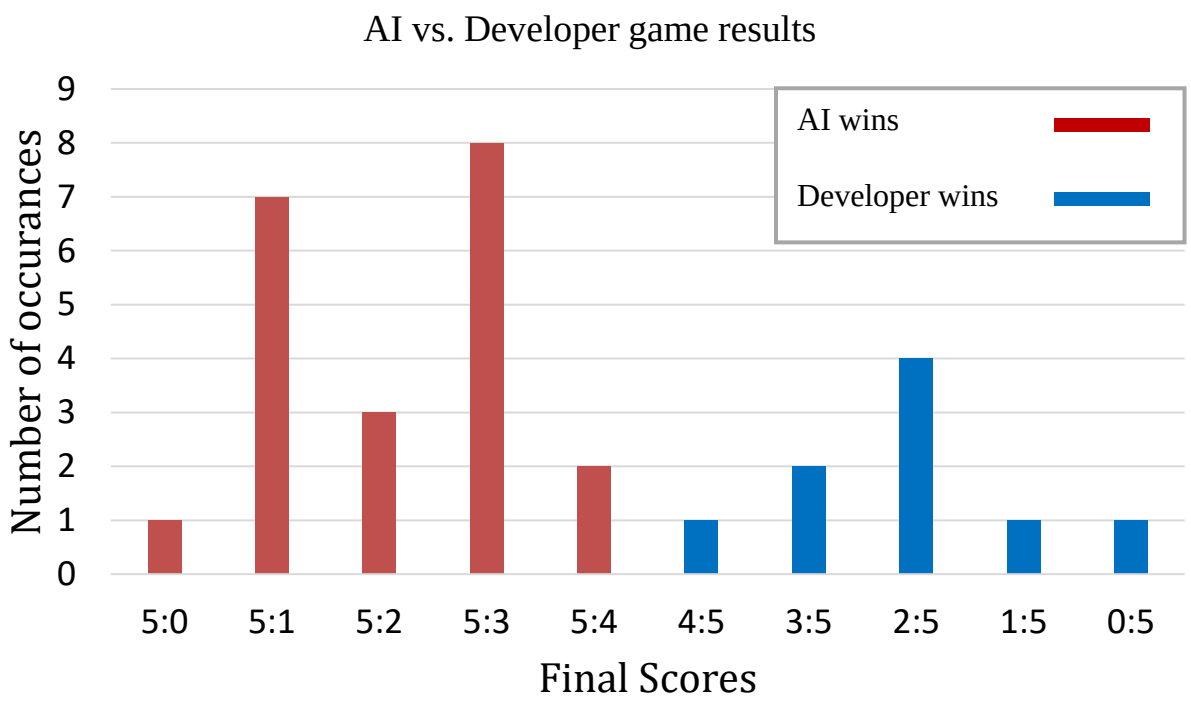


Figure 8: AI vs. Developer game results

The results of the 30 recorded matches against the developer are visible in Figure 8. These results were more balanced. Out of the 30 first-to-five games recorded by the developer, the AI won 21 games with an average lead of about 2.7 points. This amounts to a win rate of 70%. While the AI still outperformed the developer, a better handle of the game's controls did increase the odds of winning against the AI. Furthermore, the games were rather hard-won – these 30 games were played over the course of about 50 minutes. During these 30 games, a total of 214 points were scored. This means that one point took an average of 14 seconds to earn, which in the case of this game is a rather long time. Regardless, when considering these results and that the developer had a far better understanding of the controls, it is reasonable to assume that the model is competent and competitive.

When asked about the model's overall behavior not pertaining directly to its intelligence, players noted that the model had a very aggressive playstyle. When asked to select adjectives that best describe the model's playstyle, 80% of answerers selected "aggressive" as one of their two choices. When asked about exploitable weaknesses in the AI, 60% stated that they could not find any weaknesses in the AI, 30% noted that they found weaknesses but were not able to exploit them,

while 10% found weaknesses and were able to exploit them. Many players also mentioned that the AI blocked very often and considering how that visually looks (the ball shoots down towards the ground and hits the ground at high speed), it may explain why players specifically described the AI as aggressive. Thankfully, its overly aggressive playstyle did not sour the experience for players too much, since 70% of the surveyed gave a positive response when asked whether the AI was fun to play against.

Overall, when discounting comments regarding the controls of the game, the AI was received rather well by players. The AI came across as life-like; it played well and posed a significant challenge not only to new players but the game's developer themself.

4.3 Easy-AI vs. Hard-AI

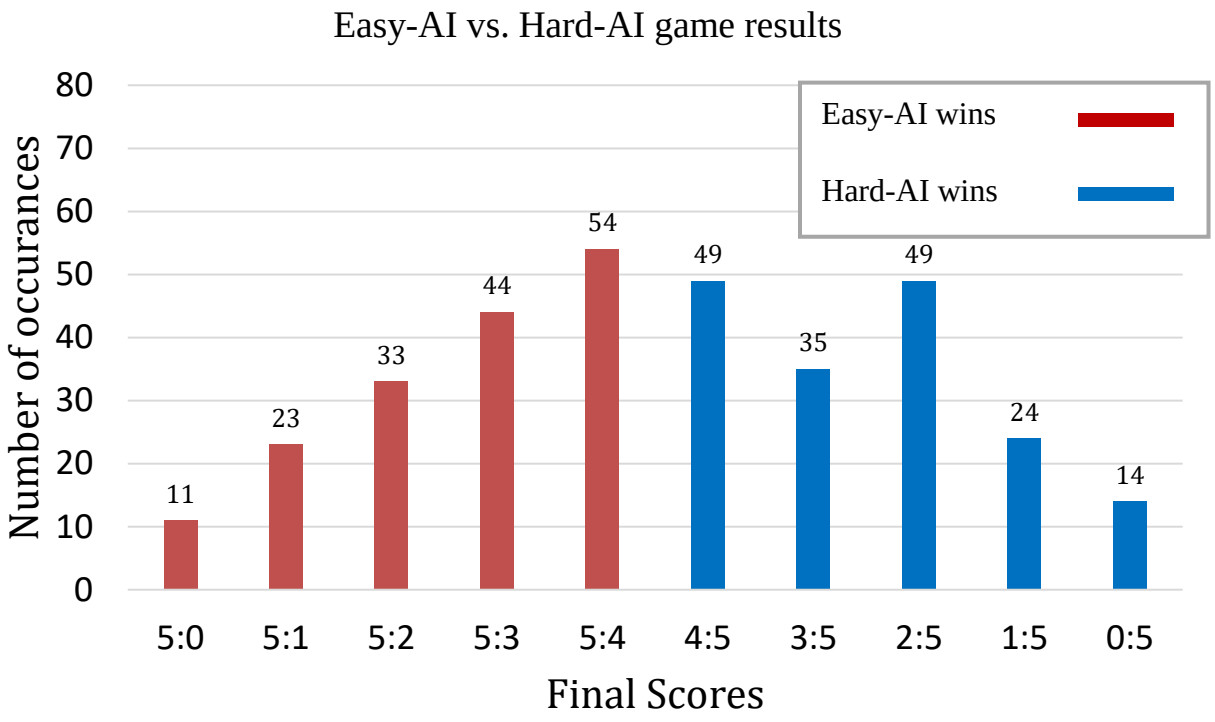


Figure 9: Easy-AI vs. Hard-AI game results

As previously mentioned, VOLL-AI has two different difficulty options for the AI, one being labeled "easy" and the other "hard." The addition of a second AI was not introduced during the initial release of the game but instead came as a post-launch update after reading feedback from players who found the original AI too challenging. Since this update came promptly after receiving feedback, the validity of the claim that one AI was easier than the other was never confirmed. To

rectify this, a test was done where the two models played a series of games against each other to see if one would prove to be a clear victor over the other.

The models played against each other for a total of 336 games and the results of these games can be seen in Figure 8 and Table 3. The difference in their performance was not as stark as initially expected, with the hard-AI winning a total of 171 games and the easy-AI a total of 165. This means the hard-AI won 50.9% of the time. This win-rate does not really corroborate the claim that one AI was more proficient than the other – both models won an almost equal amount of the times. This balance of skill level is also imminent from the distribution of different final scores – 103 games ended with a player winning with only a single point lead and games where a player won with a big lead grew less frequent as the lead increased. One interesting anomaly present in the data is how much more often the hard-AI won with a lead of 3 points as opposed to other leads. It is hard to say whether this has a concrete explanation or happened purely by chance.

When using the formula for calculating the ELO rating outlined in Section 2.5, with K = 32 and both players starting at an ELO rating of 1200, the easy-AI had a final ELO rating of 1246 while the hard-AI had an ELO rating of 1154. The easy-AI could have a better ELO rating because the easy model started winning more games at a point where the disparity between both ELO ratings was great (in favor of the hard-AI). This would increase the ELO earnings of the easy model for every subsequent win. This is supported by the fact that by calculating the ELO ratings with a shuffled order, the resulting ELO ratings changed drastically (in one instance, the final ELO ratings were 1170 for the easy-AI and 1230 for the hard-AI). The code written to calculate these values can be found in Appendix II.

4.4 Hyperparameter tuning

To test the effects of hyperparameters on the model's performance, three sets of hyperparameters were constructed and used for training. Each set of parameters was used to train a model 4 times to account for variance between different training sessions. As such, a total of 12 models were trained; however, none of these models were used in the final game – they only served a purpose for testing hyperparameters. Besides evaluating performance, this test was also important to deduce how much tuning is needed to get a model to a working state at the start of a new project.

It is also important to note that the individual effect of each parameter is not what is important in this analysis, rather how much the parameters as a whole affect the model performance.

The first set of hyperparameters consisted of the default values provided by the ML-Agents package. The second set were hyperparameters taken from the ML-Agents Tennis example, due to that project having a lot of similarities to this thesis' game. The final set was tuned based on domain knowledge of this specific thesis project. The specific contents of each hyperparameter set can be seen in Appendix III.

For the purposes of discussion, a "volley length" refers to the amount of time it takes for either player to score a point in the game. Once a point has been gained, a new volley is started. Figure 10 shows the correlation between volley length and the amount of time the model has been training for measured in the internal training steps the model has taken. For reference, 5 million training steps is about 12 hours of training. As was earlier discussed, an increase in game length may not necessarily mean better model performance, but in the case of VOLL-AI, this correlation does hold. Each line represents the mean volley length taken from the four models trained with each hyperparameter set.

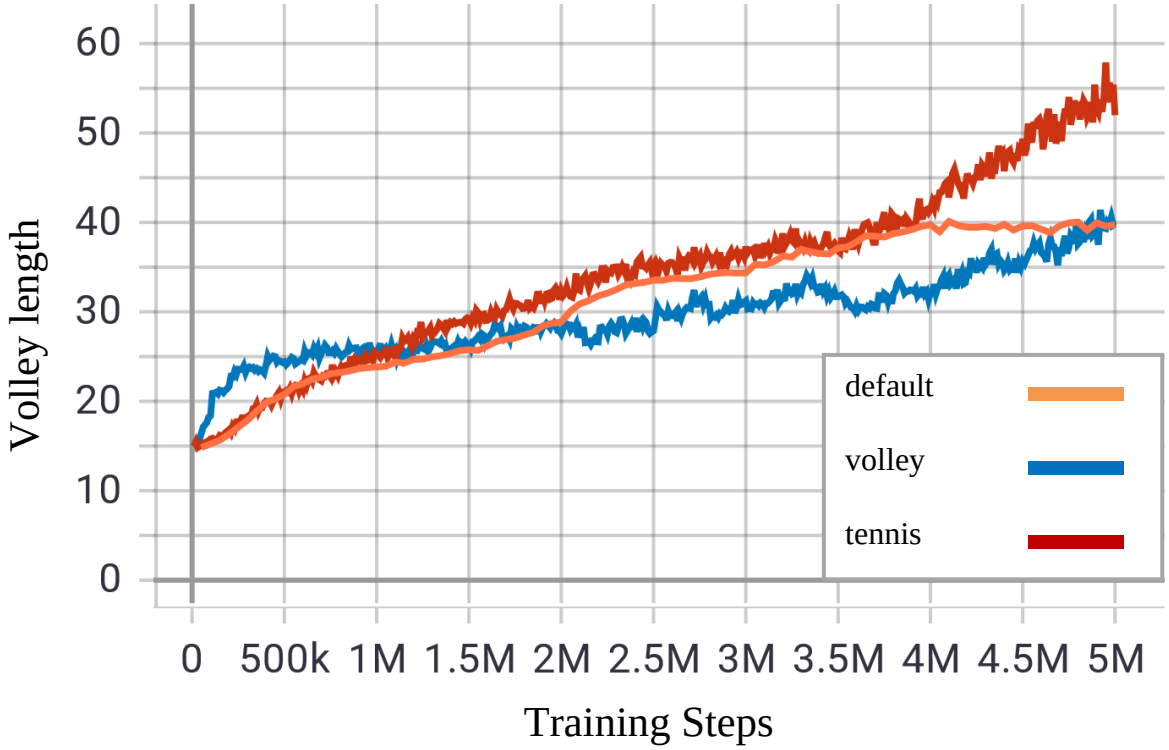


Figure 10: Volley length in relation to training steps taken.

From the graph, it is apparent that if the hyperparameters for training are tuned well, then the model's performance increase is notable. In this case, the hyperparameters taken from the ML-Agents' tennis example showed the quickest increase in episode length. This means that the volleyball and tennis games were similar enough for the same set of parameters to work for both games. The domain-specific "volley" hyperparameter set did not perform as expected, probably due to a lack of thorough testing of domain knowledge. Nevertheless, both the "volley" and "default" hyperparameter sets still displayed an increase in performance.

When watching the models trained in this experiment play the game, they displayed a minimal amount of intelligent behavior. The "default" and "volley" models were only able to learn to hit the ball once at the start of each volley, while the "tennis" models were able to serve the ball over the net as well as receive the ball from an opponent's attack. It is reasonable to assume that the models would continue improving their playstyle if they were given more time to train given the upward course of each line in Figure 10.

One interesting aspect to note was that the models did not learn different aspects of the game equally. The paragraph prior mentioned that the "default" and "volley" models were not great at serving the ball. However, both models showcased a far greater proficiency in blocking the ball, which, as noted before, is a very strong strategy. This knowledge has implications for training weaker models for a competitive game, since it shows that even with a shorter training period, a model may not be equally "weak" in every aspect of the game, which is an important thing to consider from a game design and user experience perspective.

The most notable takeaway from this test was that even with the default hyperparameters provided by Unity or otherwise poorly selected hyperparameters such as the "volley" set, the model still showed an increase in performance. This means that for some arbitrary machine-learning project akin to this thesis' work, if the models failed to learn the mechanics of a game, it is more likely that the issue is with the developer's implementation of the ML-Agents learning package and not with the hyperparameters.

Conclusion

For this thesis, a competitive volleyball game called VOLL-AI was developed and published. The game's AI was developed using Unity's ML-Agents package, which allowed the use of reinforcement learning and self-play. Developing VOLL-AI using Unity was never too difficult from a technical standpoint and most roadblocks were caused by the game's design. One of the most notable verdicts that arose from development was that when the machine learning agent is not improving, it is more likely that the issue is rooted in the developer's implementation of the ML-Agents package and not in the selected hyperparameters.

The thesis also highlighted some interesting challenges at the intersection of game development and machine learning. When developing a game that will use machine learning for artificial intelligence, it is important to consider how that affects both the design of the game as well as the technical implementation of that design. For VOLL-AI, the aspect that was most influenced by the addition of machine learning was the complexity of the player's action space and the implementation of the input interface. To improve training time and help models learn the game, aspects of how the player moves were simplified – instead of variable movement speed, the player could either move at their maximum speed or not at all, and instead of a separate button for performing the dash action, the functionality was combined with the jump button. The input interface was abstracted to ensure that the way a human player and a machine learning model interact with the game were identical, as well as to futureproof the codebase for any future changes to the design.

Based on the conducted survey, the game and its AI got a positive response on release, save for the controls of the game. The game ultimately fulfilled the goal of being playable on its own, having a distinct visual identity, containing a main menu, a tutorial as well as the ability to play against an AI or another local human player. The AI proved to be a very adept adversary to even seasoned players, having an aggressive playstyle and appearing life-like in its behavior. The aspect of the game that could have been improved upon the most were the controls, since they hindered players' enjoyment of the game and made the results of the conducted survey less reliable in determining the model's performance. The AI difficulty levels could also be improved by making them more distinct from each other in terms of play proficiency.

References

A. Aubret, L. Matignon and S. Hassas. (2019). A survey on intrinsic motivation in reinforcement learning. Univ Lyon, Université Lyon 1, CNRS, LIRIS, F-69622, Villeurbanne, France. arXiv:1908.06976v2. https://arxiv.org/abs/1908.06976.

A. Cohen. (2020). Training intelligent adversaries using self-play with ML-Agents. Unity 3D Blog. Viewed 09.04.2021. https://blogs.unity3d.com/2020/02/28/training-intelligent-adversaries-using-self-play-with-ml-agents/

A. Juliani, V. Berges, E. Teng, A. Cohen and others. (2018). Unity: A General Platform for Intelligent Agents. 07.09.2018. Viewed 15.01.2021. arXiv:1809.02627. https://github.com/Unity-Technologies/ml-agents.

B. F. Skinner. (1938). The behavior of organisms: an experimental analysis. Appleton-Century.

C. Bishop. (2006). Pattern Recognition and Machine Learning. Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA. http://users.isr.ist.utl.pt/~wurmd/Livros/school/Bishop%20-%20Pattern%20Recognition%20And%20Machine%20Learning%20-%20Springer%20%202006.pdf

D. Aversa. (2019). Ranking Systems 02 – The Elo Rating System. 22.20.2019. Viewed 18.04.2020. https://www.davideaversa.it/blog/ranking-system-2-elo-rating-system/

D. Silver and D. Hassabis. (2017). AlphaGo Zero: Starting from scratch. DeepMind 2017. https://deepmind.com/blog/article/alphago-zero-starting-scratch

E. Callaway. (2020). 'It will change everything': DeepMind's AI makes gigantic leap in solving protein structures. *Nature*. Viewed 24.01.2021. https://www.nature.com/articles/d41586-020-03348-4.

J. Hall, U. Stickler, C. Herodotou and I. Iacovides. (2020). Expressivity of creativity and creative design considerations in digital games. Computers in Human Behavior, Volume 105, 106206, ISSN 0747-5632. https://doi.org/10.1016/j.chb.2019.106206.

J. Jumper, R. Evans, A. Pritzel, T. Green and others. (2020). High Accuracy Protein Structure Prediction Using Deep Learning. 30.11.2020. In Fourteenth Critical Assessment of Techniques for Protein Structure Prediction (Abstract Book).

K. Crowley and R. S. Siegler. (1993). Flexible Strategy Use in Young Children's Tic-Tat-Toe. Carnegie Mellon University. Cognitive Science 17, 541-561, 1993. https://onlinelibrary.wiley.com/doi/pdf/10.1207/s15516709cog1704_3

K. Thirunavukkarasu, A. S. Singh, P. Rai and S. Gupta. (2018). Classification of IRIS Dataset using Classification Based KNN Algorithm in Supervised Learning. *2018 4th International Conference on Computing Communication and Automation*, Greater Noida, India. 10.1109/CCAA.2018.8777643. https://www.researchgate.net/publication/334765777_Classification_of_IRIS_Dataset_using_Classification_Based_KNN_Algorithm_in_Supervised_Learning.

Lucas N. Ferreira, Levi H. S. Lelis and Jim Whitehead. (2020). Computer-Generated Music for Tabletop Role-Playing Games. Department of Computational Media, University of California, Santa Cruz, USA. Department of Computing Science, Alberta Machine Intelligence Institute (Amii), University of Alberta. arXiv:2008.07009. https://arxiv.org/pdf/2008.07009.pdf.

M. Charytanowicz, J. Niewczas, P. Kulczycki, P. A. Kowalski and others. (2010). Complete Gradient Clustering Algorithm for Features Analysis of X-ray Images. Springer. Information Technologies in Biomedicine. Vol. 2, pages 15-24. https://www.researchgate.net/publication/226738117_Complete_Gradient_Clustering_Algorithm_for_Features_Analysis_of_X-Ray_Images

M. Minsky. (1961). Steps Toward Artificial Intelligence. Proceedings of the IRE 1961. https://courses.csail.mit.edu/6.803/pdf/steps.pdf

Richard S. Sutton andrew G. Barto. (2018). Reinforcement Learning: An Introduction. Second Edition. MIT Press, Cambridge, MA. http://incompleteideas.net/book/the-book.html.

Sebastian Heinz. (2019). Using Reinforcement Learning to play Super Mario Bros on NES using TensorFlow. *Towards Data Science*. Viewed 15.01.2021.

https://towardsdatascience.com/using-reinforcement-learning-to-play-super-mario-bros-on-nes-using-tensorflow-31281e35825.

X. Zhu, C. Vondrick, C. Fowlkes and D. Ramanan. (2016). Do We Need More Training Data? International Journal of Computer Vision 119, 76–92. arXiv:1503.01508. https://doi.org/10.1007/s11263-015-0812-2.

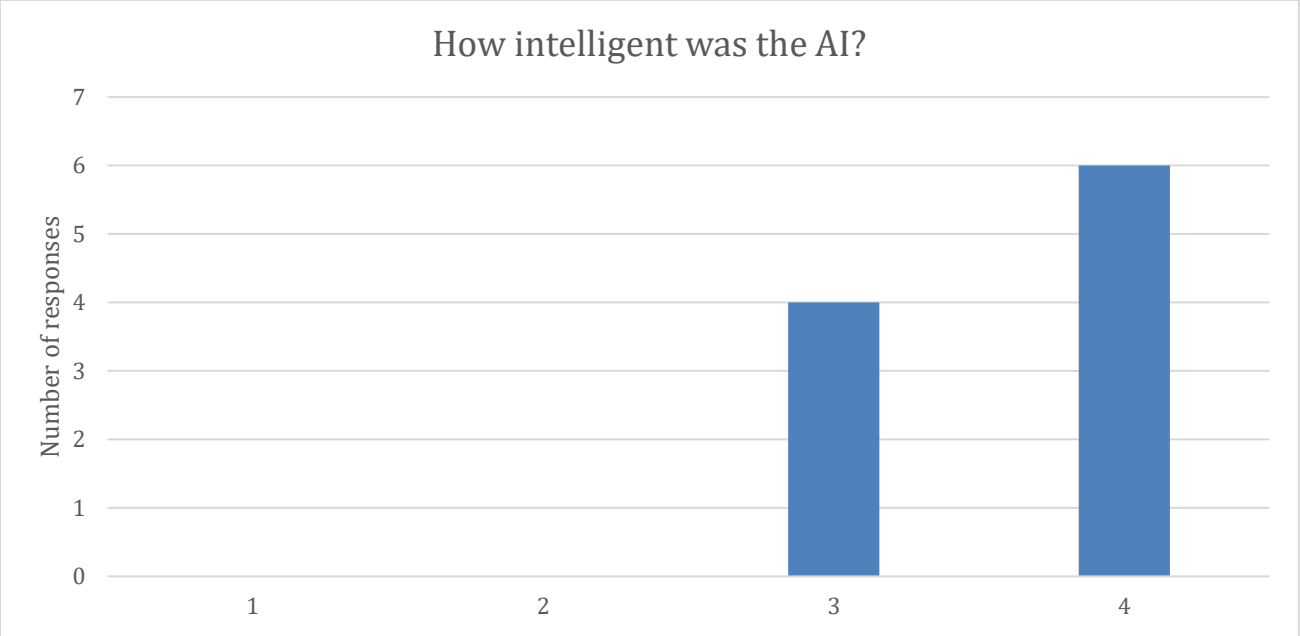
Y. Burda, H. Edwards, D. Pathak, A. Storkey and others. (2018). Large-Scale Study of Curiosity-Driven Learning. OpenAI. University of California, Berkeley. arXiv:1808.04355v1. https://arxiv.org/abs/1808.04355

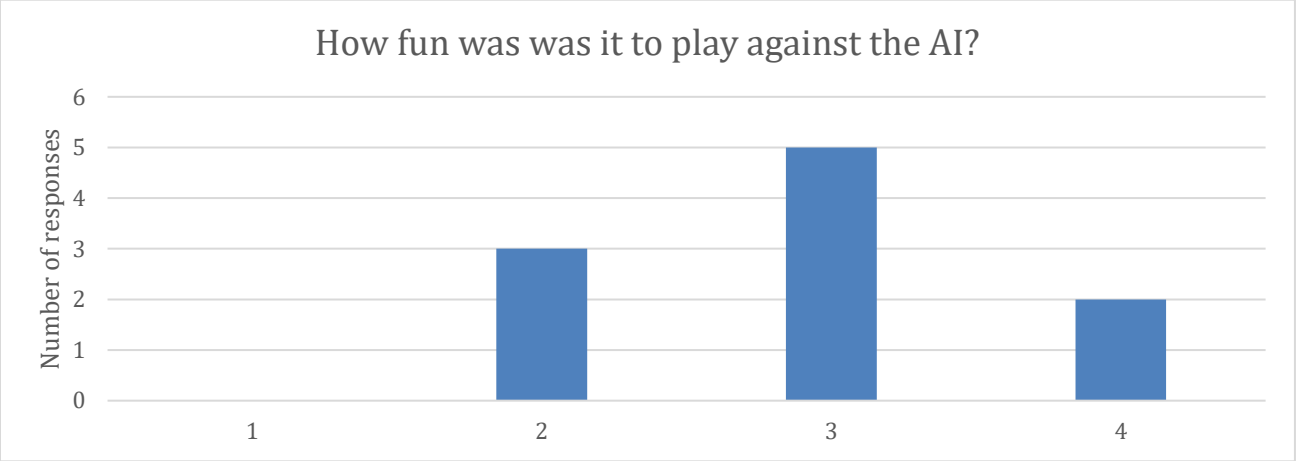
Z. Ghahramani. (2004). Unsupervised Learning. In: Bousquet O., von Luxburg U., Rätsch G. (eds) Advanced Lectures on Machine Learning. ML 2003. Lecture Notes in Computer Science, vol 3176. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-28650-9_5.

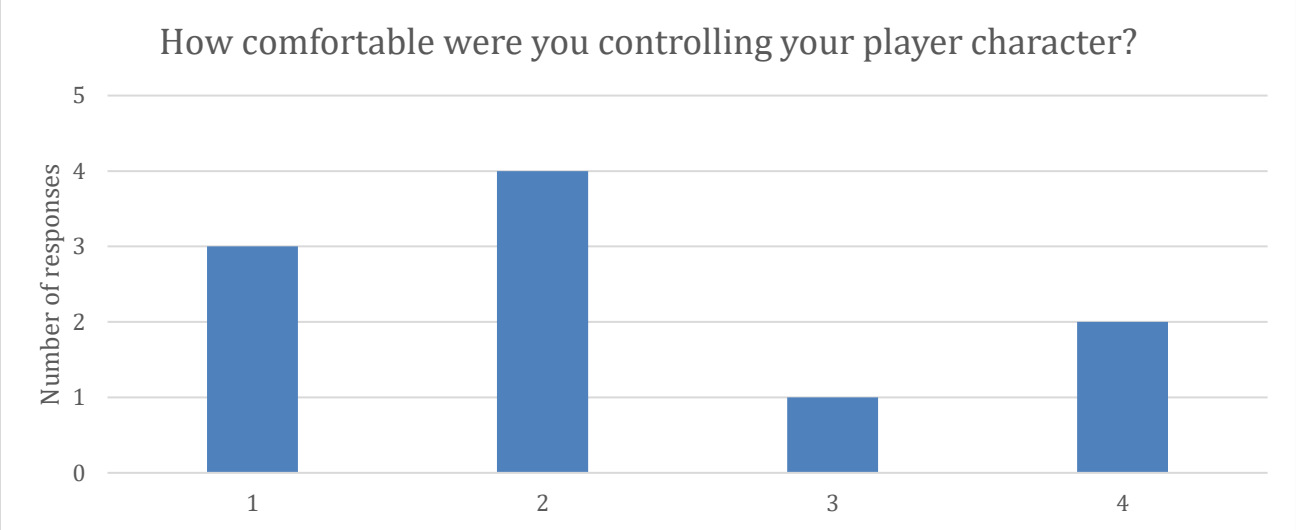
Appendix

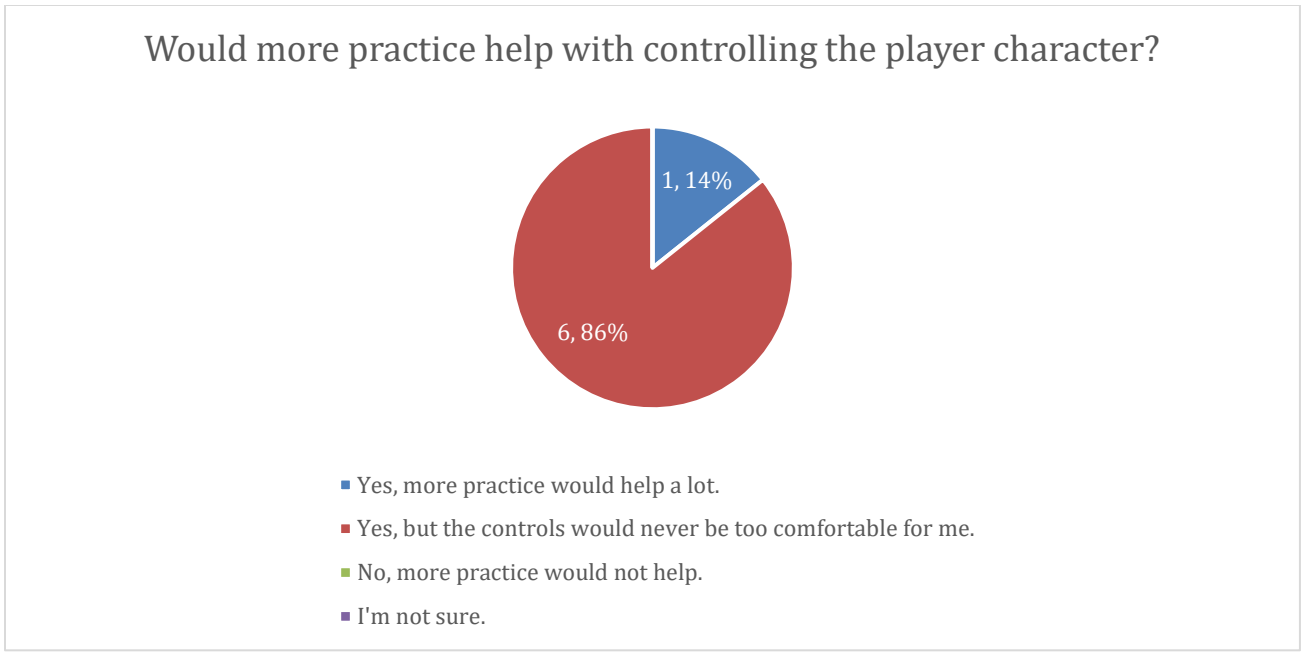
I. Survey results

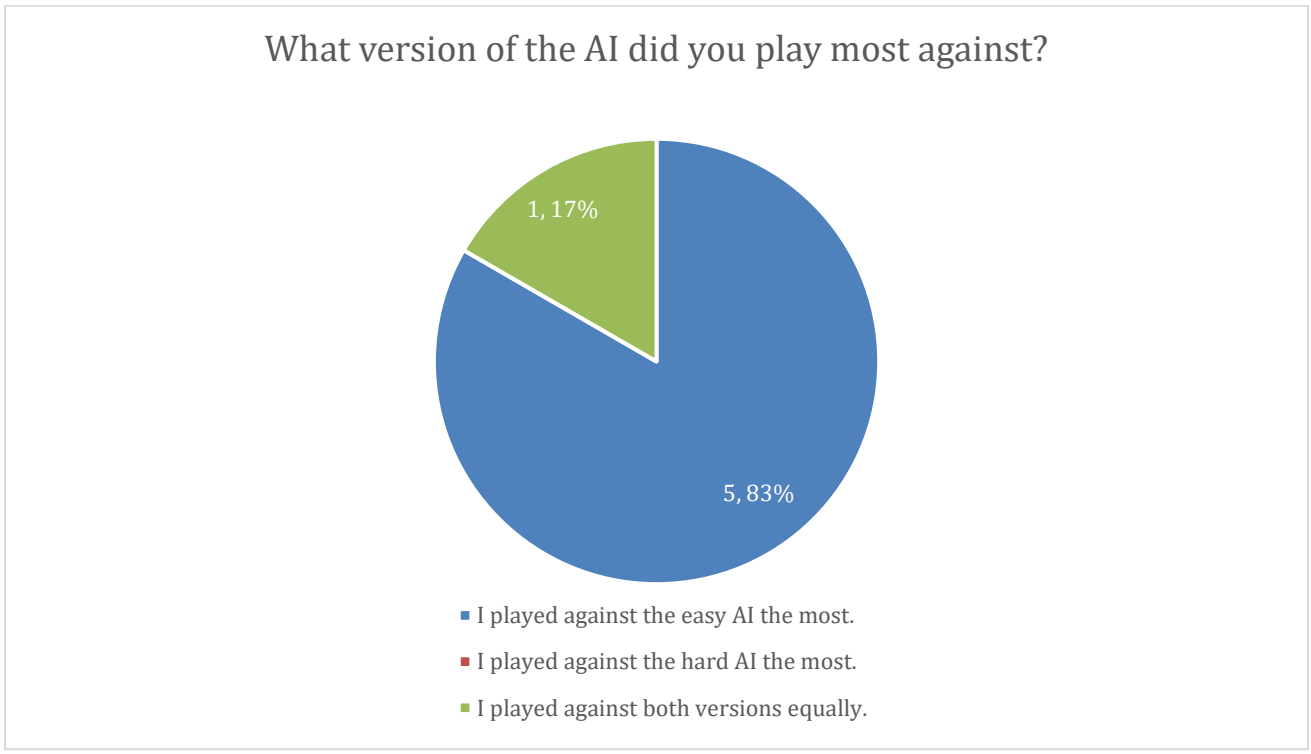
Below are the 11 questions posed in the thesis' survey. 9 questions have associated graphs, but the final 2 questions were open-ended and thus do not have a visual representation here.

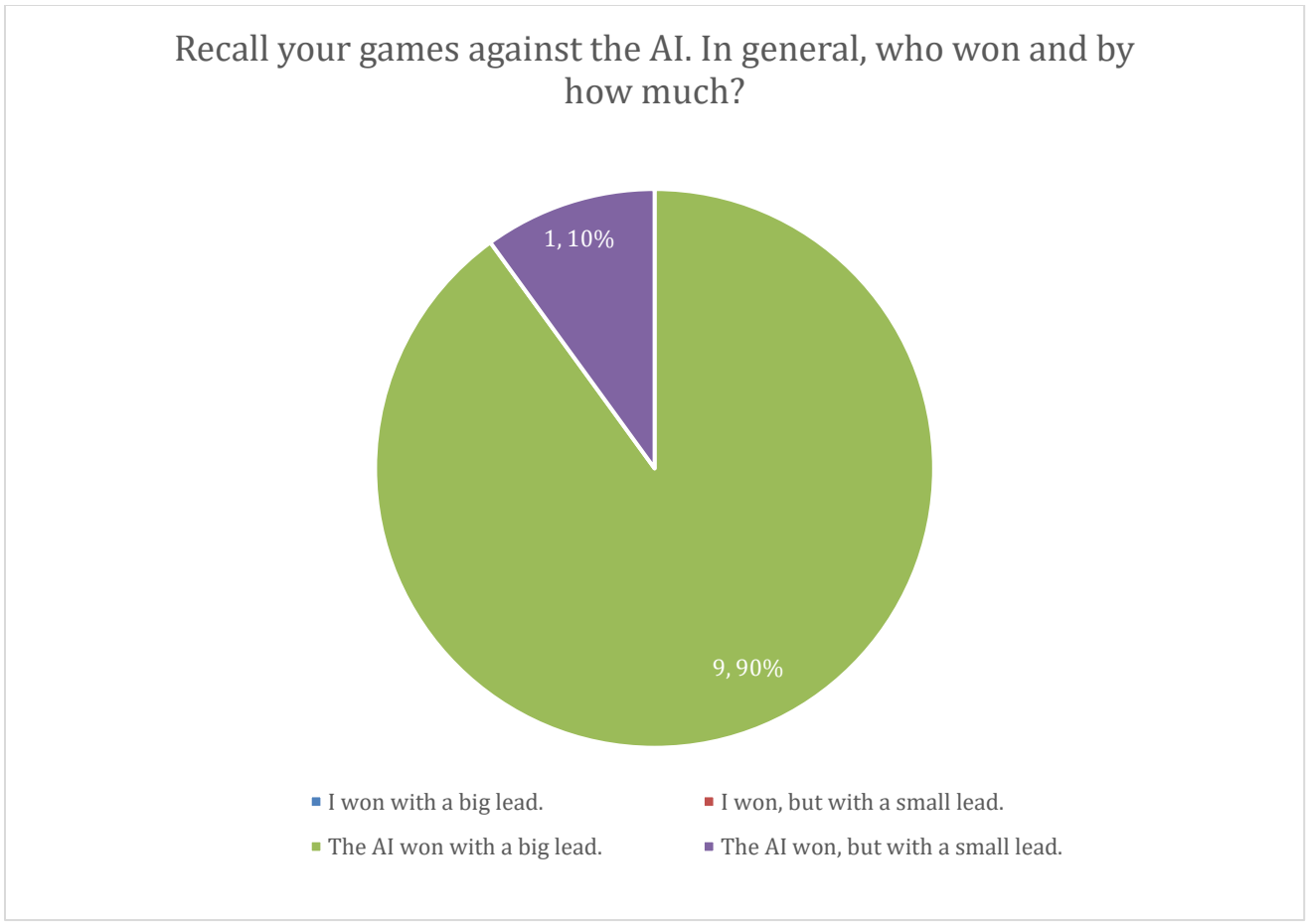

How intelligent was the AI?
Number of responses
0
1
2
3
4
5
6
7
1
2
3
4

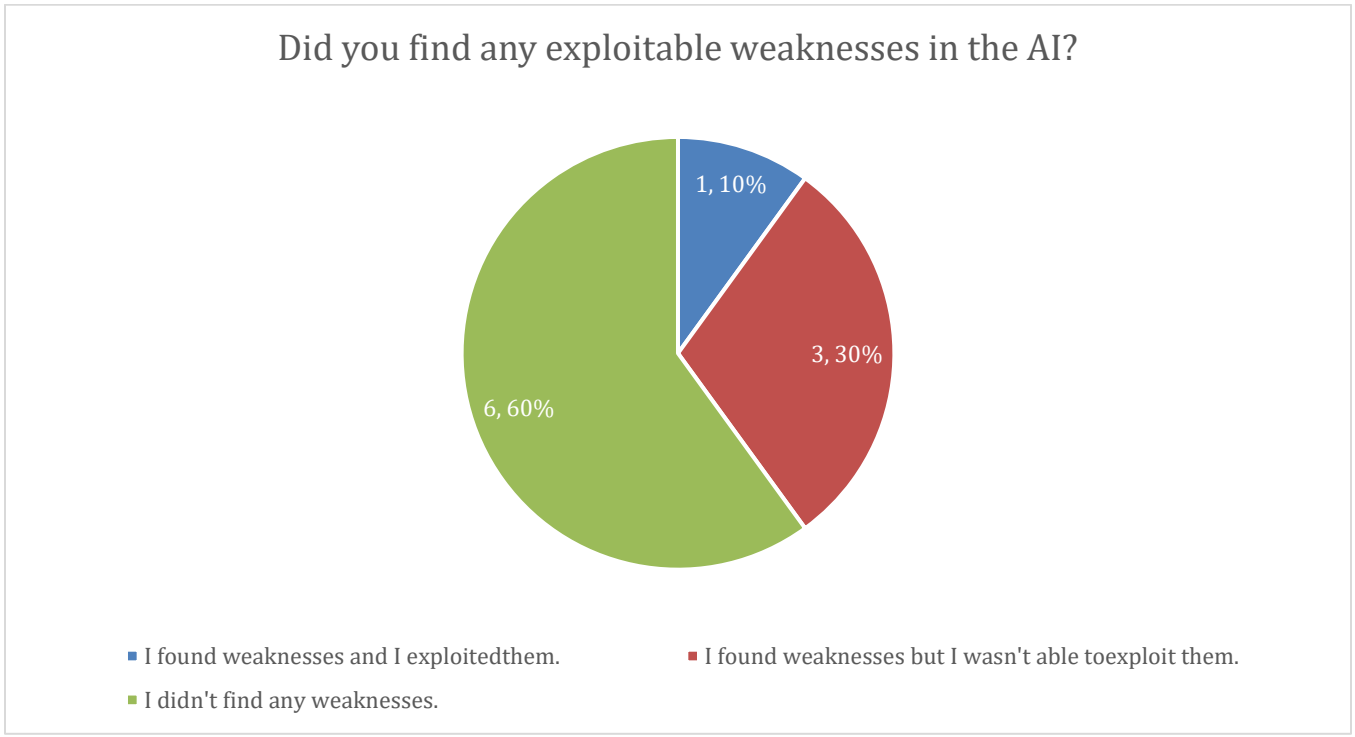

How fun was was it to play against the AI?
Number of responses
0
1
2
3
4
5
6
1
2
3
4

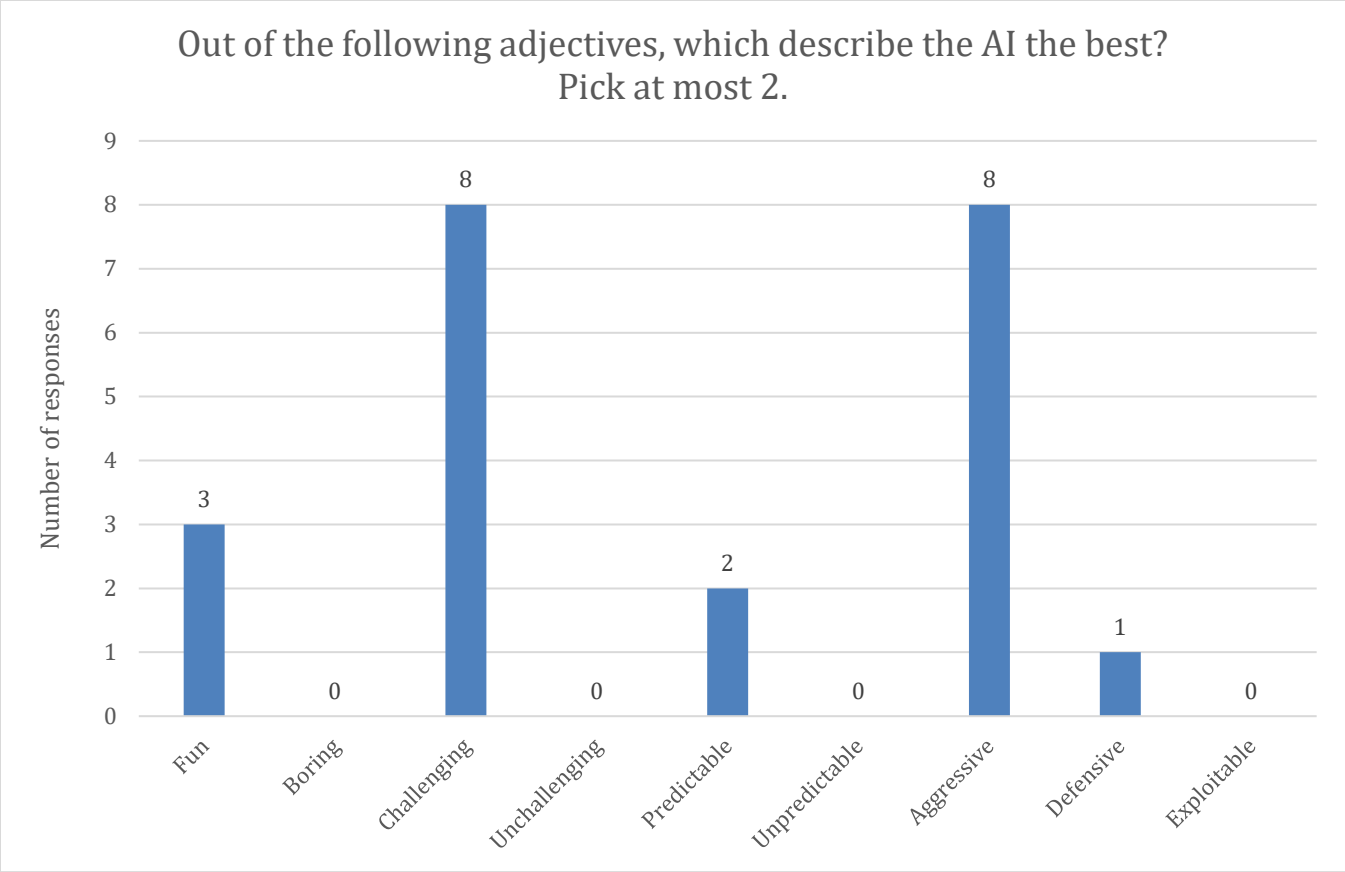

How comfortable were you controlling your player character?
Number of responses
0
1
2
3
4
5
1
2
3
4


Would more practice help with controlling the player character?
1, 14%
6, 86%
Yes, more practice would help a lot.
Yes, but the controls would never be too comfortable for me.
No, more practice would not help.
I'm not sure.


What version of the AI did you play most against?
1, 17%
5, 83%
I played against the easy AI the most.
I played against the hard AI the most.
I played against both versions equally.


Recall your games against the AI. In general, who won and by how much?
1, 10%
9, 90%
I won with a big lead.
I won, but with a small lead.
The AI won with a big lead.
The AI won, but with a small lead.


Did you find any exploitable weaknesses in the AI?
1, 10%
3, 30%
6, 60%
I found weaknesses and I exploitedthem.
I found weaknesses but I wasn't able toexploit them.
I didn't find any weaknesses.



How did the AI influence your style of play? How did that make you feel?

I tried my best to beat them but whatever I did, I could not. This made me a little sad, however, I understand this is what they are supposed to do.
I don't think it is fair to compare an incompetent child to a proficeint volleyballer. I wonder if somebody who played volleyball actively would have been better at the game.
Its aggressive attacking play made me try to jump to block. It wasn't successful and I felt frustrated.
Due to its aggressive playstyle i picked defensive strategy, it didn't help though :c
I played more defensively. On hard mode, the AI kept absolutely dunking on me.
It made me play more defensively then I was expecting. I felt scared

It made me put more effort in when trying to defeat it. However, the main challenge was that the game was too fast for me and my reflexes were too slow. Which is why I also suck at real life volleyball :D
It really felt like playing against a person!
i was desparately trying to get a point, which made me stressed but determined. It was fun that after a couple of games of practice i was able to get points in, even though the ai still beat me handedly.

If you have any other thoughts regarding the AI, please share them here.

Not related to AI! I enjoyed the mechanics of the game, the minimalism and cohesion. It was well designed.
maybe make difficulty levels for AI? In the current state its is too strong
It will take over the world, it already has mastered volley ball.
Its a hard game but i think a change to the controlls could change a lot of that percived difficulty.

II. ELO script

The code used to calculate ELO ratings in Section 4.3.

```
import pandas as pd
file_name = 'points.csv'
results = pd.read_csv(file_name)
left_elo = 1200
right_elo = 1200
k = 32
count = 0

# Used as reference:
# https://metinmediamath.wordpress.com/2013/11/27/how-to-calculate-the-elo-
rating-including-example/
for index, row in results.sample(frac=1).iterrows():
  left_transformed = pow(10, left_elo/400)
  right_transformed = pow(10, right_elo/400)

  left_expected = left_transformed / (left_transformed + right_transformed)
  right_expected = right_transformed / (left_transformed + right_transformed)

  left_won = 1 if row[0] == 5 else 0
  right_won = 0 if row[0] == 5 else 1
  count = count + right_won

  left_elo = left_elo + k * (left_won - left_expected)
  right_elo = right_elo + k * (right_won - right_expected)

print(round(left_elo), round(right_elo))
```

III. Hyperparameter sets

Below are the specific hyperparameters used for training the three variations of models in Section 4.4.

```
behaviors:
 Volleyball:
  trainer_type: ppo
  max_steps: 5000000
  self_play:
   save_steps: 20000
```

Default hyperparameters provided by Unity ML-Agents.

```
behaviors:
 Volleyball:
  trainer_type: ppo
  hyperparameters:
   batch_size: 2048
   buffer_size: 20480
   learning_rate: 0.0003
   beta: 0.005
   epsilon: 0.2
   lambd: 0.95
   num_epoch: 3
   learning_rate_schedule: constant
  network_settings:
   normalize: true
   hidden_units: 256
   num_layers: 2
   vis_encode_type: simple
  reward_signals:
   extrinsic:
    gamma: 0.99
    strength: 1.0
  keep_checkpoints: 5
  max_steps: 5000000
  time_horizon: 1000
  summary_freq: 10000
  threaded: true
  self_play:
   save_steps: 50000
   team_change: 100000
   swap_steps: 2000
   window: 10
   play_against_latest_model_ratio: 0.5
   initial_elo: 1200.0
```

Parameters used in the ML-Agents Tennis example.

```
behaviors:
 Volleyball:
  trainer_type: ppo
  hyperparameters:
   batch_size: 32
   buffer_size: 2048
   learning_rate: 0.0001
   beta: 0.005
   epsilon: 0.2
   lambd: 0.95
   num_epoch: 24
   learning_rate_schedule: linear
  network_settings:
   normalize: false
   hidden_units: 128
   num_layers: 2
   vis_encode_type: simple
  reward_signals:
   extrinsic:
    gamma: 0.995
    strength: 1.0
  max_steps: 5000000
  time_horizon: 32
  summary_freq: 10000
  threaded: true
  self_play:
   save_steps: 20000
   team_change: 100000
   swap_steps: 100000
   window: 10
   play_against_latest_model_ratio: 0.5
   initial_elo: 1200.0
```

Hyperparameters tuned using domain knowledge of this thesis' game.

The default hyperparameter set seems short, but that is because ML-Agents applies the default values if the respective parameters are absent from the configuration file. Only the minimal required parameters were written up.

The parameter "max_steps" was give the same value in each hyperparameter set. This was done to ensure that each model trained the same amount of time and a more valid comparison could be draw between each trained model.

IV. License

Non-exclusive licence to reproduce thesis and make thesis public

I, Tanel Marran,

1. herewith grant the University of Tartu a free permit (non-exclusive license) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, *Developing a Volleyball Game with an AI Opponent Using Reinforcement Learning*, supervised by Tambet Matiisen.
2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons license CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. I certify that granting the non-exclusive license does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Tanel Marran

30/05/2021