TRAINING ARTIFICIAL INTELLIGENCE AGENTS TO PLAY A TOWER DEFENSE GAME USING REINFORCEMENT LEARNING

Mitchell Harrison, Jonathan Rivera, Justin Stevens, Dr. Chad Hogg Millersville University
{mdharri2, jjriver1, jmsteve1, chad.hogg}@millersville.edu

ABSTRACT

Tower Defense games have captured a wide audience, drawing them in with their strategic complexity and challenging gameplay. Players must strategically place defensive towers and make quick decisions to fend off waves of enemies. Harnessing the potential of Reinforcement Learning (RL) algorithms offers a promising avenue for discovering optimal strategies through iterative learning. This paper explains the implementation of a tower defense game in Unity and our experiences in training a reinforcement learning model to play optimally.

1. Introduction

Bloons Tower Defense (BTD), a game series in the tower defense genre, is known for its blend of strategy and fastpaced action. Our project aims to recreate this game within a Unity environment and design an artificial intelligence (AI) to play the game. Recreating the game instead of interfacing directly with BTD allows us to design a more complex environment for the AI. This AI will utilize a reinforcement learning (RL) algorithm known as Proximal Policy Optimization (PPO) that will be expanded upon later.

While other AI algorithms could be used, we utilized PPO as it is a newer RL algorithm that fixes many common issues faced with traditional RL algorithms [1]. These issues most notably include sparse reward environments, which is a case for our game necessitating the use of PPO or related algorithms that could address this issue.

Our goal for this AI is to have it play the game that we recreate in the same manner as a human and eventually be able to play at or above the level of a human. However, as we will see later, designing an AI to play a game this complex is far from a trivial task and requires a significant amount of parameter tweaking and experimenting.

2. Building A Test Environment

Tower defense is a popular genre in the world of video games, requiring players to strategically place defensive structures, known as towers, to prevent waves of enemies from advancing toward a designated point on the map. One notable example within this genre is a game series named Bloons Tower Defense (BTD), where players face off against colorful balloon enemies known as "bloons" and must use various monkey towers equipped with different weapons and abilities to pop them before they reach the end of the path. This classic game series serves as the basis for our project.

2.1 Background: Bloons Tower Defense

Figure 1: Progression of bloon types.

In Bloons Tower Defense (BTD), bloons are categorized by color, each representing varying levels of health and speed. Health refers to how many hits it takes to "pop" or destroy a bloon, while speed refers to how quickly a bloon moves along the path. The progression of bloon colors, as illustrated in Figure 1, represents an increase in both health and speed. It begins with red, the weakest and slowest bloon, and progresses to blue, green, yellow, pink, white, and finally, black, the strongest and fastest type.

When a higher-level bloon is popped, it is replaced by the next lowest-level. For example, a popped green reveals a blue that when popped reveals a red that when popped is removed from the game. Thus, high level bloons require multiple sequential attacks to completely destroy.

The game progresses through waves, or "rounds," with each wave introducing increasingly challenging bloon formations. As players advance through the waves, they must deal with a higher number of bloons, as well as bloons with greater health and speed.

Players can earn in-game currency by popping bloons and completing rounds, with the amount earned directly linked to the number of bloons popped and the current round number. This currency serves as the primary resource for purchasing towers and upgrades. In addition to managing currency, players must also monitor their "lives" throughout the game. Lives represent the player's health and are depleted each time a bloon successfully reaches the designated point on the map, with the game ending once all lives are gone. Higher-level bloons cost more lives when they are "leaked".

To defend against bloons, players must use defensive towers called monkeys. The monkey towers possess a variety of different abilities and upgrades each costing a different amount of money. Players must strategically choose which towers to place, where to place them, and how to upgrade them to effectively pop bloons and progress through the game. In the BTD game series, players can choose from over 24 distinct types of monkeys. However, for our project, we focused on implementing the dart monkey, boomerang monkey, cannon tower, tack tower, ninja monkey, super monkey, dartling gunner, and sniper monkey.

2.2 Background: Unity Game Engine

While brainstorming for the project, we came to a dilemma on which environment to use to train the AI. We could either train the AI in any of the six commercially released Bloons Tower Defense games, but this would require an interface for the AI to interact with the game and way to retrieve important game data and supply it to the AI. Or we could recreate the game from scratch and have the interface built into the game for the AI to interact with directly. We deemed the latter to be an easier and more suitable approach since we could then tweak the game to suit our needs better and could make a more interesting environment for the AI. For this endeavor of recreating the game, we decided on the Unity Engine due to the team's prior experience with the engine.

The Unity Engine is a popular 2D and 3D game engine that supports modern game development features to aid in development [2]. It has an easier entry barrier than other popular engines such as Unreal Engine, due to its use of C# as the main scripting engine compared to C++, at the cost of performance which is mostly negligible in a 2D environment that the Bloons game requires [2].

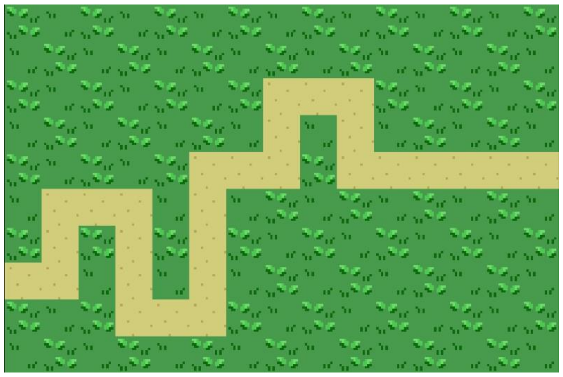


Figure 2: A randomly generated 10 by 16 map in our game.

2.3 Map Generation

Unlike the actual BTD games that offer a variety of prebuilt maps to choose from, ranging from easy to hard difficulty, we opted to have our map randomly generated for each game. Figure 2 shows an example of a randomlygenerated map. Between each playthrough the path of the bloons and the tower placement areas are different, making the process of training an AI to play the game in general much more interesting. However, the rules that remain constant when randomly generating a map are that the path must always go from the left side to the right side without ever entering the top or bottom row and any two non-consecutive path tiles cannot be touching. Both decisions were made to avoid confusion from human players.

The map is displayed as a series of tiles that form a grid. A grass tile was used to represent a spot where a player could place a tower, and a sand tile was used to denote the path of the bloons where the player could not place towers. The path of the bloons is a series of connected tiles that the bloons must follow, shown in Figure 2. Once the map was generated on screen, we used a 2D array to track the state of the map, allowing us to provide important environment details to the AI later.

2.4 Wave Manager

The Wave Manager script is a crucial part of managing the waves of enemies in the game. It helps determine when waves start, how enemies are spawned, and when waves end. This script allows for customizable settings like the cooldown time between waves to give the player time to alter their defensive measures and whether the waves are randomly generated or not. One key feature of the Wave Manager is its ability to semi-randomly generate waves based on predefined difficulty levels called RBEs (Red Bloon Equivalents). These RBEs represent the difficulty of a wave in a numerical form. For example, the red bloon has an RBE of 1 since it has one HP, the blue bloon has 2 HP, so it has an RBE of 2 and so forth. So, for any given wave the RBE for that wave is predetermined and a random selection of bloons is chosen such that the sum of the RBEs of the chosen bloons will be equal to the wave RBE.

The script selects which types of enemies to spawn in each wave from a list of possible enemies. This list of possible enemies is updated as the player progresses through the waves with a hardcoded table, as we did not want one of the hardest bloons to spawn in on an early wave, mimicking the behavior of the actual BTD games. For example, on wave 1 it is only possible for the red bloon to spawn. Then we add the blue bloon on wave 2, the green bloon on wave 6, etc. Another important aspect is how it handles the progression of waves. It starts waves either manually, with player input, or automatically after a predefined number of seconds, whichever comes first. It keeps track of the current wave number and checks if a wave is completed by monitoring the number of enemies on the map and the number left to spawn.

The difference between our Wave Manager system and the BTD games is the approach to wave generation. In our system, waves are semi-randomly generated based on predefined difficulty levels (RBEs), allowing for variation in enemy types and quantities. Conversely, BTD games employ waves made for each level until going past the level limit, with specific enemy types and spawn patterns designed for a progressively challenging experience.

After this level limit then it becomes more like ours with semi random bloons spawning with an infinite level cap. Another distinction is in wave progression control since we allow flexibility in starting waves manually or automatically, with customizable time intervals between waves. In contrast, BTD games tightly control wave progression within the level design, with predetermined intervals and scripted events aligning with the game's difficulty curve.

Additionally, the Wave Manager organizes enemies into groups based on the bloon type/color, each with its own spawn interval and number of enemies to spawn. This allows for more control over how enemies are distributed throughout each wave. However, the Non-Random Wave Manager script offers a different approach to managing waves. Instead of semi-random generation, it relies on predefined wave events. Each wave event contains specific information about the types and quantities of enemies to spawn, as well as the timing of their spawn. These wave events provide a more scripted and controlled experience compared to semi-random generation. They allow developers to design waves with precise timing and enemy compositions, leading to more tailored gameplay experiences.

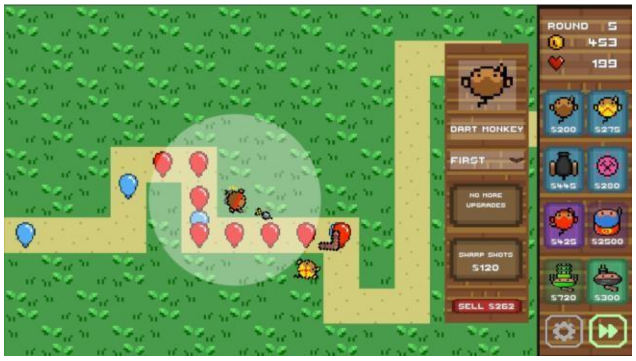


Figure 3: In game screenshot of a randomly generated map on wave 5 with blue and red bloons on the path. The dart monkey is currently highlighted with its one upgrade path finished and is currently firing at a red bloon. A yellow boomerang monkey is also shown near the dart monkey.

2.5 Tower Implementation

In our tower defense game, the functionality of our towers is defined in a script known as the monkey script. Each tower has its own specific tower script that inherits from the generic monkey script to avoid duplication of generic monkey behavior.

Within the monkey script, we defined variables that each tower will have, such as the tower's cost to purchase, firing rate, targeting mode, projectile speed, and the number of bloon layers it can pop per hit. Additionally, the script contains methods that control the tower's behavior and interactions with the game environment. When the script is called, it continuously checks for nearby bloon enemies within the tower's range. Based on the tower's targeting mode, the script then selects an appropriate target. Once the target has been chosen, the tower fires a projectile at the enemy, dealing damage based on variables described previously.

The monkey script also defines abstract methods for tower upgrades, providing a framework for other tower scripts to override and implement custom upgrade logic that is tailored to each tower type. Towers typically have two upgrade paths, each consisting of two upgrades. These upgrade paths enable towers to enhance their capabilities, such as increasing firing rate, projectile damage, or unlocking new abilities.

Using the framework established by the monkey script, we proceeded to implement eight distinct tower types. These towers include the dart monkey (shown in Figure 3), boomerang monkey, cannon tower, tack tower, ninja monkey, super monkey, dartling gunner, and sniper monkey. Among these, the dart monkey and sniper monkey became particularly important due to their usefulness in training our AI.

The dart monkey is set up to be an average tower in our game, like BTD, with an average projectile speed. It can pop one bloon layer per hit, making it effective against red bloons. Through its first upgrade path, the dart monkey can improve its range to reach further targets, while its second upgrade path increases its popping power, allowing it to pop up to four bloon layers per hit. The dart monkey's average range and projectile speed limit its effectiveness against harder bloon types.

The sniper monkey was set up to be a longer-range tower with certain advantages over the dart monkey. Its projectile speed is significantly faster, which enables it to pop faster-moving bloons. Also, it has a range 25 times larger than that of the dart monkey, covering the entire map. To compensate, the sniper monkey fires much less frequently than the dart monkey. While it can initially pop the same number of bloon layers per hit as the dart monkey, the sniper monkey's first upgrade path can increase this number from one to seven, meaning that it can destroy most of the higher tier bloons in one hit.
Alongside this upgrade, its second upgrade path focuses on increasing the projectile speed. The sniper monkey's ability to pop bloons from a greater distance and with increased speed makes it a powerful asset in popping faster-moving bloons during later rounds of the game.

3. Building An Intelligent Agent

Artificial Intelligence (AI) refers to the act of intelligently completing actions to achieve an end goal by a computer entity. An action refers to a task being completed that affects an environment that the AI is acting upon [3]. For our project, we used a specific field of AI called Reinforcement Learning (RL). RL classed agents, an instance of an AI, learn to interact with an environment by taking actions to maximize cumulative rewards. These rewards are defined by the user and are given to the agent when a decision is to be made or when a goal is or is not achieved [3]. However, it is worth noting that these rewards can be both positive or negative, positive denoting the agent made a series of good decisions and negative denoting a series of bad decisions.

Through repeated trial and error, the agent refines its decision-making process, often employing policy networks and gradient-based mathematical methods to optimize its actions. However, basic RL encounters challenges such as difficulty in effectively providing negative rewards for undesirable behavior and dealing with sparse or dense reward structures [4]. To address these issues, techniques such as modifying reward structures to encourage desirable behavior have been developed, improving the effectiveness and efficiency of RL algorithms. Despite this, there can still be issues with the policy gradient. The gradient update occurring could overfit or underfit, or the function might not be able to comprehend the output so it chooses a suboptimal action that it can explain [4]. These issues can be addressed however with Open AI's Proximal Policy Optimization (PPO) algorithms.

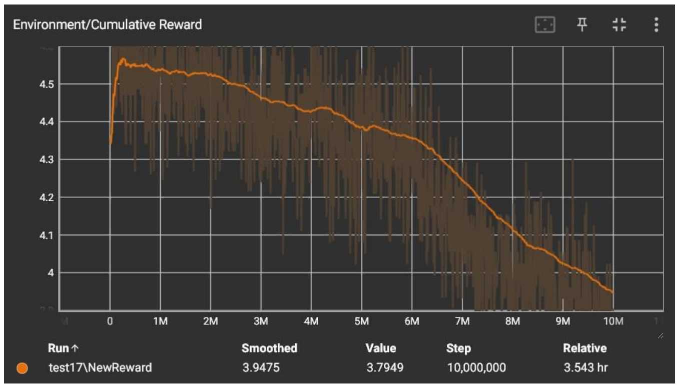
3.1 Background: Proximal Policy Optimization

PPO stands out as a prominent algorithm in the realm of RL, acclaimed for its stability and efficacy in training deep neural network policies [5]. As a member of the policy gradient methods family, PPO directly optimizes policies to maximize expected cumulative rewards. Distinguished by its on-policy approach, PPO learns from experiences generated by the latest iteration of its policy, ensuring that the training data remains relevant and circumventing issues associated with data distribution mismatches encountered in off-policy methods.

Furthermore, PPO integrates supplementary terms into its loss functions, including updates to the baseline estimator (LtVF) and entropy regularization (SΠθ), along with weighting coefficients (c1 and c2) found in Figure 4. These components collectively augment training stability and performance across a diverse array of RL tasks, contributing to PPO's resilience and effectiveness called hyperparameters. Hyperparameters are used in shaping the performance and behavior of the PPO algorithm. Parameters such as the entropy coefficient profoundly influence the training dynamics and convergence properties of PPO.

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right],$$

Figure 4: The full formula used in PPO algorithm.



Run ↑	Smoothed	Value	Step	Relative
● test17\NewReward	3.9475	3.7949	10,000,000	3.543 hr

Figure 5: Graph of cumulative reward over time during a training session for an AI model with an overly complex environment. The training session was over 3.5 hours. Bright line shows smoothed value with the darker lines showing the actual value. The X axis shows the number of steps or decisions, and the Y axis shows the average cumulative reward per round.

3.2 Background: ML Agents Library

The goal of this project was primarily focused on the applications of RL, specifically PPO, in BTD and for this reason, we decided to use ML-Agents, a Unity Engine plugin that already supported PPO and other RL algorithms. It is characterized as a general-purpose machine learning agents toolkit that allows for games and simulation to serve as environments for training intelligent agents. [6] It uses the Python Pytorch and TensorFlow libraries on the backend and provides functionality to allow the programmer to interact with the libraries at a high level within Unity [6].

To use the package, the programmer needs only to choose an RL model such as PPO, define the parameters for the model with a configuration file, provide observations to the model, receive, and carry out actions, and define a reward system. Once an environment is completed and contains an agent from the library, the user simply starts up a Pytorch environment from the command line and starts the Unity game allowing the two to connect over a device port and the agent will begin training for as many simulations defined by the agent's configuration file.

During training sessions, the Python environment collects data at predetermined intervals defined in the configuration file and aggregates this data into a JSON file, also providing a way to view this data in the form of a webpage to analyze the AI during and after training in a more human-friendly way.

3.3 Agent Configuration Overview

The main purpose of the AI that we were designing was to reach as high of a wave as possible. To give it the best possible chance of doing this, we first allowed the AI to place most of the towers, play on a full-sized map, and be able to have full control over the towers (upgrade, buy, and sell). However, we quickly noticed the AI was learning very slowly or even becoming worse, in terms of cumulative reward as shown in Figure 5. We were also dealing with limited computing resources and time, so we decided to simplify the game for the AI. The plan was now to train the AI in a simple environment with limited actions then slowly allow the AI to have more actions and give it a larger environment after we determined it was successful at its current stage. Eventually the goal is to have the AI trained and efficiently playing in the full environment that we designed with zero restrictions on its actions.

3.4 Agent Configuration: Environment

Originally, we designed the AI training environment to be the same as the player environment, meaning that the AI was playing on a 10 by 16 grid map. However, with such a large map we were not seeing the AI learn anything during its training of about 10-15 hours. To speed up its training, we decided to train the AI on a smaller map and through numerous training scenarios. We determined that a 6 by 10 map was both simple enough to train the AI on, but still large enough where tower placement and other strategies still mattered.

Another concern with the environment was that originally, only one AI agent could play a game at a time. The MLAgents package offered support for training multiple agents using the same model, and we wanted to utilize that. So, we built an environment that had 51 agents each playing their own game at the same time. However, allowing multiple agents to train in the same environment led to many issues that we did not foresee. All these issues stemmed from using static variables in the scripts to ease development. For more context, since we had at least 51 instances of each script, we could no longer use any static variables and had to redesign scripts that utilized those. After completing this redesign and allowing the AI to train, we immediately noticed that the number of completed episodes, or waves in our case, within the same time period was higher, leading to faster training times.

3.5 Agent Configuration: Observations

For the agent to be able to take actions, it needed to know about its environment using observations. The number of observations that we could provide to the model was fixed per training session due to the nature of ML-Agents and PPO. Due to this, we decided to only provide the model with what we deemed the most important observations that the agent would need to make intelligent decisions. These observations were the 60 tile states that came from the 6 by 10 map, the agent's current amount of money, and the agent's current wave, all of which were normalized as recommended by ML-Agents. We decided not to give it information on the bloons at the current state, as the number of bloons was frequently changing and the number of observations must stay the same number. However, we do plan on fixing this shortcoming and addressing it later.

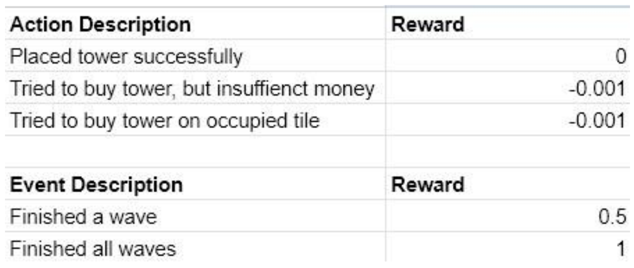
To represent each tile state for the AI, we used enumerations to define a tile state associated with a number. For example, we used a 0 to denote an open tile that could have a tower placed on, 1 for a bloon path tile, 2 for a tile that contains a dart monkey, and 3 for a tile that contains a sniper monkey. As we implemented more monkeys for the AI to utilize, we would continue the pattern.

3.6 Agent Configuration: Actions

When an agent is ready to take an action as defined by the configuration file for the model, it makes 4 primary decisions. The first primary decision is what the agent wants to do (nothing, buy tower, sell tower, or upgrade), the second is the tower type that it wants to buy, the third is the tile location of interest, and the last is the targeting mode for the tower it is about to place. For our current version, it can only choose to do nothing or buy a tower, buy a dart monkey or a sniper monkey, choose any of the 60 tiles from the 6 by 10 map, and choose any of the three targeting modes currently available (first, last, or strongest).

Defining the actions in this way with ML-Agents leads to an issue: the package only allows the user to block the agent from taking a specific decision, but not a series of decisions in the same action. For example, we can block the agent from choosing to sell a tower, but we cannot block it from choosing to sell a tower and choosing the first targeting mode. The latter action does not make sense in relation to our game as when you are selling a tower it does not make sense to also choose a targeting mode, however we cannot prevent this.

For the current model, the one type of action that we could block was the AI from buying a tower unless it had enough money, since the action of buying a tower was just one primary action. We decided to do this, so we could limit the number of times that the AI would receive a negative reward.



Action Description	Reward
Placed tower successfully	0
Tried to buy tower, but insuffienct money	-0.001
Tried to buy tower on occupied tile	-0.001
Event Description	**Reward**
Finished a wave	0.5
Finished all waves	1

Figure 6: The current reward system for the AI.

3.7 Agent Configuration: Reward System

To allow an AI agent to train and effectively learn an appropriate reward system must be defined for the model. The reward system dictates when and how the agent receives a reward, whether the reward should be positive or negative, and the weight of the reward. The agent then keeps track of its cumulative reward and the model attempts to maximize its reward during training while also exploring different series of actions. However, as defined by the configuration file for the AI model, the agent will begin to explore less over time and focus more on maximizing the reward with what it has already learned [6].

While experimenting with our AI agent, we quickly found that giving the agent too many negative rewards led to its learning quickly becoming unstable, or in other words its cumulative reward would drop over time as shown in figure 5. We later learned that ML-Agents suggests giving the positive rewards much more frequently and of a higher weight than the negative rewards [6], which led us to the reward system shown in Figure 6. This system was tweaked over the course of over 25 training simulations and still has room for improvement.

As you can see in Figure 6, the positive rewards are of a much higher weight than the negative rewards. This was needed as the agent will finish a wave much less frequently than it will try to buy a tower or place one on an occupied tile. This is because the agent makes roughly one decision per second and an entire wave can take between 30 to 120 seconds.

3.8 Results

Our most recent training iteration of the AI took place over 5.5 days, with over 230,000 rounds played in over 22,000 games. The aggregated data with graphs is shown in Figures 7 through 10. Looking at Figure 7, we can see that the cumulative reward is gradually increasing during the AI training. Whereas previously, the cumulative reward dropped over time, as shown in Figure 5. We suspect redefining the reward system with fewer penalties is what caused an increase in cumulative reward.

Looking at Figure 10, it can be seen that the average wave reached drops slightly at the beginning; however, at the very end of the training, the average wave reaches its highest point. This supports the previous idea that the AI is doing better over time by going from an average of about wave 8 or 9 to almost 11.

One may begin to question: if the average wave went up by 2, with a reward value of 0.5, how did the cumulative reward increase to 4 points? This can be explained by the AI placing towers in valid spots more frequently and avoiding buying towers when it does not have enough money. This led to the AI receiving many fewer penalties and achieving a much higher reward over time.

In Figure 11, there is a much higher number of dart monkeys compared to sniper monkeys. This is also supported by Figure 8, where it shows an average of about 1 sniper monkey being bought per game and the frequency staying roughly the same. Whereas in Figure 9, the agent slowly buys more dart monkeys over time, with an average of about 4.5 per game.

Initially, we suspected that the AI deemed the sniper monkey to be a weaker tower compared to the dart monkey. However, from our own playthroughs, we knew that this wasn't true, as the sniper monkey has a much farther range and can target any bloon on the map. Whereas the dart monkey needed strategic placement to ensure that it could cover a good portion of the bloon path. Also, when looking back at previous training data, we found that the AI preferred the sniper monkey more.

One change from earlier training iterations was to prevent agents from purchasing towers unless they possessed sufficient funds to do so. Since the dart monkey cost $100 less than the sniper monkey, as soon as the action became unblocked for buying a dart monkey, it instantly bought that tower and never waited to buy a sniper monkey. The one time it had enough money to buy a sniper monkey was at the very start of the game since the agents start with $500. This is why in Figure 11, there is only 1 sniper monkey and many more dart monkeys. In future iterations, we plan to either get rid of this action block or find appropriate hyperparameters that would encourage it to explore more possible actions, like buying the sniper monkey.

4. Related Work

We are neither the first people to be interested in developing a tower defense game of our own, nor the first to try to train an agent to intelligent play such games. A Colombian team developed a domain-specific language for modeling tower defense games in 2015. [7]. In 2019, another group of authors explored procedural generation of tower defense maps, using a much more sophisticated version of our map generator. [8] Another group approached this problem through genetic algorithms. [9] Work on random wave generation to produce a flow state in players was released in 2022. [10]

Even more work has been published in the area of artificial intelligence for playing tower defense games. Some have proposed the use of optimization models. [11] Others use genetic algorithms to refine neural networks that make tower-placement choices. [12] Most relevant to our work, some use deep reinforcement learning based on neural networks. Our work uses a different type of learning algorithm and different conceptions of actions, observations, and rewards than these prior works. [13, 14]

5. Future Work

When creating future iterations of our game, we would want the AI to have access to a broader array of towers and game mechanics. Currently, the AI is limited to utilizing only the dart monkey and the sniper monkey. We aim to expand its capabilities by allowing it to interact with additional towers such as the boomerang monkey, ninja monkey, tack tower, super monkey, cannon tower, and dartling gunner. We also plan on allowing the AI to upgrade its towers and sell them, which would give the AI more choices and allow for more interesting emergent behavior. Additionally, equipping the AI with information about the bloons would benefit the AI in terms of further complexity as it currently lacks awareness of their existence due to them not being an observation. This would make selling towers a viable option as it could relocate towers to a more effective area based on bloon information. The AI could also learn other behavior based on the bloon information that we have not predicted.

We would also redefine our hyperparameter tuning methods. In previous iterations, the hyperparameter tuning we conducted was very sporadic, as we did not keep a system of parameter changes and their corresponding effects on the AI's performance. We did not keep track of these changes as we were changing a variety of hyperparameters, rewards, and observations between a training session due to a lack of time and computing resources. We intend to adopt a more structured approach by modifying one parameter at a time, thereby gaining a clearer understanding of the impact of each adjustment on the AI's behavior and rewards.

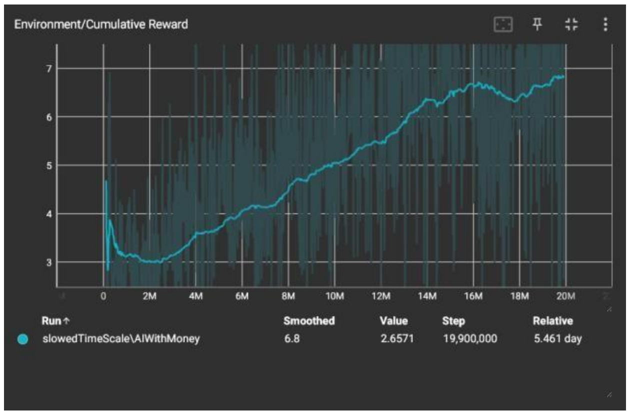
Once the AI's access to game elements and its tuning process have been optimized, we plan to have further extensive training sessions. By lengthening the duration of training sessions, the AI will have a larger observation field to learn from. With this the AI becomes more proficient at learning. This could involve adding additional upgrades, towers, monkeys, and other elements from BTD into our game.

6. Conclusion

Our research aimed to train AI agents for a tower defense game using reinforcement learning. Despite challenges, our project showcased promising advancements. Initially, the AI struggled to learn efficiently due to complex environments and limited computing resources. However, by simplifying the game environment and redesigning scripts to accommodate multiple agents, we accelerated training and observed a gradual improvement in cumulative rewards.

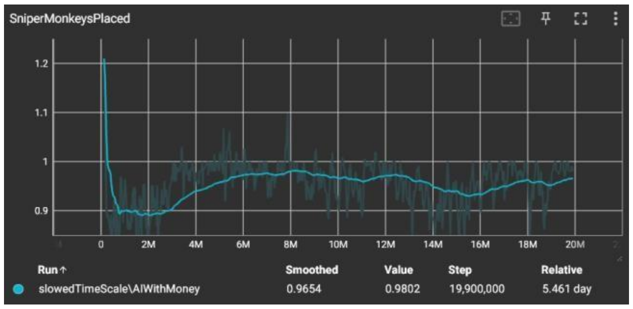
Key adjustments, such as refining the reward system to prioritize positive rewards and limiting the AI's access to towers and game mechanics in the short term, enhance performance over time. Additionally, a structured approach to hyperparameter tuning and extended training sessions are planned to further optimize the AI's capabilities. Our project lays the foundation for future advancements in AI-driven tower defense gameplay, offering insights into the potential of reinforcement learning algorithms in gaming environments.

Figures



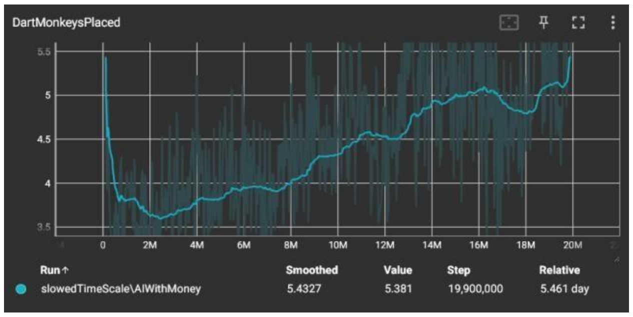
Run ↑	Smoothed	Value	Step	Relative
● slowedTimeScale\AIWithMoney	6.8	2.6571	19,900,000	5.461 day

Figure 7: Graph of the average cumulative reward per episode for the most recent iteration of the AI agent.



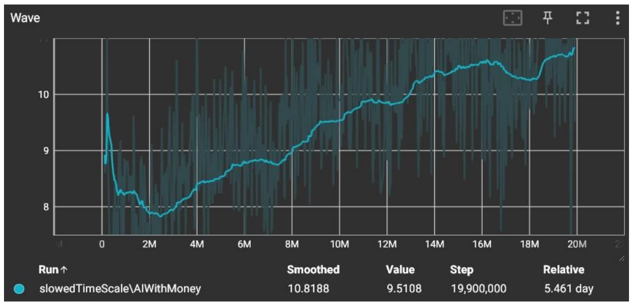
Run ↑	Smoothed	Value	Step	Relative
● slowedTimeScale\AIWithMoney	0.9654	0.9802	19,900,000	5.461 day

Figure 8: Graph of the average number of sniper monkeys placed per episode on the Y axis for the most recent iteration of the AI agent. The X axis shows the number of steps or decisions.



Run ↑	Smoothed	Value	Step	Relative
● slowedTimeScale\AIWithMoney	5.4327	5.381	19,900,000	5.461 day

Figure 9: Graph of the average number of dart monkeys placed per episode on the Y axis for the most recent iteration of the AI agent. The X axis shows the number of steps or decisions.



Run ↑	Smoothed	Value	Step	Relative
● slowedTimeScale\AIWithMoney	10.8188	9.5108	19,900,000	5.461 day

Figure 10: Graph of the average wave reached per episode on the Y axis for the most recent iteration of the AI agent. The X axis shows the number of steps or decisions.

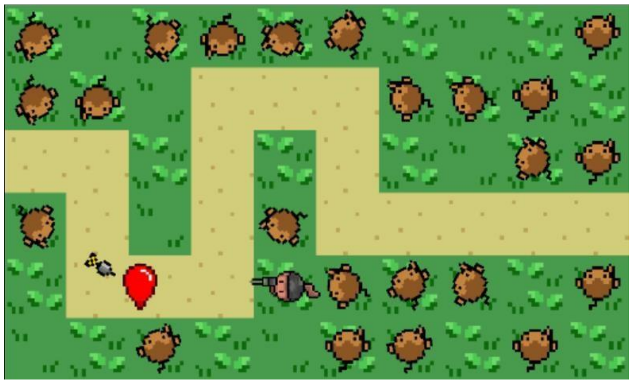


Figure 11: In game screenshot of the most recent AI iteration of the AI agent playing, currently on wave 22.

References:

[1] OpenAI. 2024. OpenAI Baselines: Proximal Policy Optimization. Retrieved February 24, 2024 from https://openai.com/research/openai-baselines-ppo

[2] Unity Real-Time Development Platform. (n.d.). Retrieved from https://unity.com/

[3] J. Shin, T. A. Badgwell, K.-H. Liu, and J. H. Lee. 2019. "Reinforcement Learning – Overview of recent progress and implications for process control. Retrieved February 24, 2024 from https://www.sciencedirect.com/science/article/abs/pii/S0098135419300754

[4] J. Hare. 2019. Dealing with Sparse Rewards in Reinforcement Learning. Retrieved February 24, 2024 from https://arxiv.org/abs/1910.09281

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. 2024. Proximal Policy Optimization Algorithms. Retrieved February 24, 2024 from https://arxiv.org/abs/1707.06347

[6] Machine learning agents. (n.d.). Retrieved from https://unity.com/products/machine-learning-agents

[7] K. Sanchez, K. Garces, & R. Casallas, A DSL for rapid prototyping of cross-platform tower defense games. In Proc. 10th Computing Colombian Conf., Bogota, CO, 2015, 93-99.

[8] S. Liu, et. al., Automatic generation of tower defense levels using PCG. In Proc. 14th Intl. Conf. on the Foundations of Digital Games, Sann Luis Obispo, CA, USA, 2019, 1-9.

[9] V. Kraner, I. Fister Jr., & L. Brezocnik, Procedural Content Generation of Custom Tower Defense Game using Genetic Algorithms. In Proc. 4th New Technologies, Development and Applications, Sarajevo, BA, 2021, 493503.

[10] D. Hind & C. Harvey, A NEAT Approach to Wave Generation in Tower Defense Games. In Proc. Intl. Conf. On Interactive Media, Smart Systems, and Emerging Technologies, Limassol, CY, 2022, 1-8.

[11] O. Mazurova, O. Samantsov, O. Topchii, & M. Shirokopetleva, A Study of Optimization Models for Creation of Artificial Intelligence for the Computer Game in the Tower Defense Genre. In Proc. Intl. Conf. On Problem of Infocommunications, Kharkiv, UA, 2020, 491496.

[12] T. G. Tan, Y. N. Yong, K. O. Chin, J. Teo, & R. Alfred, Automated Evaluation for AI Controllersin Tower Defense Game Using Genetic Algorithm. In Proc. Soft Computing Applications and Intelligent Systems, Shah Alam, MY, 2013, 135-146.

[13] A. Dias, J. Foleiss, & R. P. Lopes, Reinforcement Learning in Tower Defense. In Proc. 2nd Videogame Sciences and Arts, Mirandela, PT, 2020, 127-139.

[14] A. Ramirez, Neural Networks Applied to a Tower Defense Video Game. A Thesis at Jaume I University, Castella de la Plana, ES, 2018.