Strategic Mastery in Terminal: A Novel Reinforcement Learning Approach to Tower-Defense Gameplay

Ishaan Javali (ijavali@princeton.edu)
Department of Computer Science, Princeton University
Princeton, NJ 08540 USA

Yagiz Devre (yagiz.devre@princeton.edu)
Department of Computer Science, Princeton University
Princeton, NJ 08540 USA

Shivam Kak (sk3686@princeton.edu)
Department of Computer Science, Princeton University
Princeton, NJ 08540 USA

Abstract

From chess to Atari to AlphaGo, performance in games has been a benchmark of performance for machine learning. Terminal is a tower-defense game where players submit algorithms that play against each other over a diamond-shaped board. The goal of the game is to breach the opponent's defenses and reach the border of the gameboard. Terminal has hundreds of thousands of annual players, with strategies typically involving thousands of hard-coded, rule-based, logical cases for the tower defense simulation. This is a very limited approach as the number of game board configurations is over 10^{250}!

Thus far, no Reinforcement Learning-based approaches have been attempted for Terminal. We propose a novel solution to Terminal using methods in Reinforcement Learning (RL) and machine learning that achieves a ranking among the top 300 players world-wide, placing the algorithm in the top 1

Moreover, as pioneers in the field, we have curated a publicly-available dataset of 30,000 online matches played by over 1,000 different algorithms that can be used for further research of the Terminal game for RL.

Our approach utilizes a RL technique known as behavioral cloning along with convolutional neural networks, which allows for our AI agent to analyze the game board and learn to mimic expert players from our dataset of online matches. This innovative approach not only demonstrates the effectiveness of RL in Terminal but also contributes to the broader advancement of AI in gaming. Our next steps are to improve the algorithm through thousands of games of self-play.

Keywords: Reinforcement learning, behavioral cloning, data cleaning, state space, convolutional neural network, proximal policy optimization (PPO), natural policy gradient, actor/critic networks, web scraping, trajectory-weighted reward function

Introduction

Correlation One's Terminal represents a strategic challenge set within the Tower Defense genre, where players compete in a diamond-shaped arena by deploying mobile units and structures. The game requires a balance between offensive tactics and defensive strategies, utilizing two types of resources—mobile points (MP) for attacking units and structure points (SP) for stationary defenses. This game is not only about battling an opponent but also about planning, forecasting, and real-time decision-making. The game effectively demands players to deploy mobile units and construct defensive structures using allocated resources strategically. The ultimate objective is to diminish the opponent's health points to zero, employing a mix of offensive maneuvers and defensive tactics.

This paper seeks to explore the application of Reinforcement Learning (RL) to navigate the strategic complexities of Terminal, thereby optimizing gameplay strategies beyond human capabilities. This challenge is significant because Terminal requires a deep understanding of both long-term strategy and immediate tactical responses, reflecting real-world decision-making scenarios.

The difficulty lies in the game's requirement for both strategic planning and adaptability to dynamic conditions. The state space of Terminal is vast and complex, presenting a significant challenge for traditional game-playing algorithms. Despite the popularity and strategic depth of Terminal, there has been no previous work that applies machine learning (ML) or reinforcement learning (RL) techniques to this game. Traditionally, players in Terminal have relied on extensive if-else logic and manually programmed rules to guide the deployment and positioning of units. This approach requires defining thousands of specific rules and handling numerous edge cases to cover as many game situations as possible. However, these hard-coded, rule-based methods often lacked flexibility and scalability, making them less effective against unpredictable or novel strategies employed by opponents. This gap presents a unique opportunity to explore and pioneer the use of sophisticated ML and RL methods in Terminal, potentially transforming how the game is played and setting a groundwork for future AI research in complex strategic environments.

The reinforcement learning (RL) approach offers a promising alternative to these traditional methods due to its ability to adapt and generalize from varied game situations. Unlike algorithms that depend on predefined conditions, reinforcement learning models enables dynamic strategy formation, allowing the model to make improved decisions based on the current state without the constraints of hard-coded logic. This enhanced adaptability makes RL models capable of effectively responding to a wide range of opponent tactics.

Furthermore, the game dynamics of Terminal add to the complexity, with simultaneous actions required every round, demanding both prediction capabilities and strategy adjustments according to your opponent. The state space includes not only the variety of possible unit placements and movements but also the types of units and their strategic implications depending on their positioning on the board which will be described in the following section. Moreover, the action space in Terminal is enormous due to the many potential actions a player can take each turn, influenced by the current health, budget, and the state of both the player's and the opponent's defenses.

Adding to the challenge is the absence of explicit and immediate rewards in Terminal. Success in the game is not solely about reducing the opponent's health; it also involves managing one's resources effectively and strategically positioning units to both defend and attack. This advanced combination of factors makes RL particularly suited for Terminal, as it can learn and optimize strategies based on long-term outcomes rather than immediate gains, navigating the large state and action spaces with more flexibility than traditional rule-based systems.

This project, therefore, seeks to pioneer the use of reinforcement learning to understand the strategic layers of Terminal. By leveraging advanced RL techniques, we aim to develop an algorithm that not only competes at the highest levels within the game but also provides a template for applying similar strategies in broader, real-world contexts that mirror the complex decision-making environments found in Terminal. Through this exploration, we anticipate contributing to the field of AI by demonstrating the practical application and benefits of reinforcement learning in dynamic and strategic settings, pushing the boundaries of what AI can achieve in competitive and adaptive environments.

Game Mechanics

Fundamentally, Terminal, as described earlier, is form of Tower Defense Game which makes it a perfect strategy based game to play with a Reinforcement Learning algorithm. The game itself is a two-player, simultaneous-turns game meaning that every player simultanously plays a set of actions in each round. The game involves strategic deployment of units, management of resources, and real-time decision-making, all unfolding within a competitive setting.

Game Initialization

At the start of each game in Terminal, players begin with a standardized setup that establishes the initial conditions. Each player is allotted an equal and fixed number of structure points (SP) and mobile points (MP), typically starting with 40 SP and 5 MP, which they can use to deploy their initial units and structures. Additionally, both players start with a set health total, usually 30 Health Points (HP), which they must defend while attempting to reduce their opponent's HP to zero. Finally, since no player has placed a unit, the gameboard starts as an empty arena.

Game Board

The game is laid out in a diamond grid, represented in Figure 1, with a size of 28×28. Each player controls one half of the diamond, which is mirrored across the central horizontal axis. One player occupies the bottom half, starting from the bottom point, while the other controls the top half from the top point. This division enforces a clear territorial distinction between the two players. At each round, both players simultanously decide and deploy 2 classes of units, defence or offense, to their own half of the board.

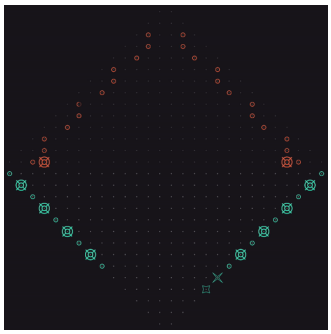

Figure 1: Game Board

Action Space

Deploy : For the 210 points that the player controls on their half of the board, the players can select any of the three Structure units for defences in each turn as long as there is a budget for such deployment:

- Wall: A cheap defence unit that prevent opponent Mobile Units. Takes 1 Structure Credits, has an HP of 60.
- Support: An expensive unit that provides shielding to friendly units that pass within a certain range(3.5 tiles). Takes 4 Structure Credits, has an HP of 30.
- Turret: An expensive unit that deals 5 damage to an enemy Mobile unit within a certain range(2.5 tiles). Takes 2 Structure Credits, has an HP of 75.

This makes the possible action space for simply deploying structure units 3^{210}! Additional to the defence units, the players can select any of the three mobile units in each turn, however the mobile units can be only place on the 28 edge points of their side of the diamond arena(14 on both sides).

- Scout: A cheap, fast-moving mobile unit that deals light damage. Takes 1 Mobile Credit, has an HP of 15 and deals 2 damage to the opponent structures.
- Demolisher: An expensive, slow-moving mobile unit that deals high damage to the opponent structures. Takes 3 Mobile Credits, has an HP of 5 and deals 8 damage to the opponent structure.

- Interceptor: A cheap, very-slow-moving unit that deals 20 damage to an enemy Mobile units within a certain range(4.5 tiles). Takes 1 Mobile Credits, has an HP of 40 and deals 20 damage to the opponent Mobile Units but does 0 damage to the opponent Structure Units.

One additional factor to consider is that players are allowed to stack multiple attacking units on the same location in the deployment phase to gain strategic advantage, since the enemy structure units target one mobile unit at a time. However, unlike the mobile units, it is not possible to stack multiple structure units.

Upgrade and Remove : Additional to deploying units, a player is able to effectively remove or upgrade a structure unit that has already been placed in the desired location, adding depth and flexibility to their defensive and offensive strategies. Upgrading a structure typically enhances its capabilities, such as increasing its health, range, or damage output, which can be critical for fortifying positions or extending control over key areas of the board. Conversely, removing a structure provides strategic adaptability, allowing players to recover a portion of the spent structure points (SP) as a refund and redeploy them more effectively in response to the shifting dynamics of the game by the following amount:

$$\text{Refund} = 0.75 \times \text{Initial Cost} \times \frac{\text{Remaining Health}}{\text{Original Health}}$$

The given unit information, therefore, can be summarized as in Table 1.

Table 1: Attributes of Attack and Defense Units in Terminal

Unit Type	Cost	Health	Damage	Range
Attack Units				
Scout	1 MP	15	2	3.5
Demolisher	3 MP	5	8	4.5
Interceptor	1 MP	40	20	4.5
Defense Units				
Wall	1 SP	60	0	0
Support	4 SP	30	0	3.5
Turret	2 SP	75	5	2.5
Upgraded Defenses				
Upgraded Wall	2 SP	**120**	0	0
Upgraded Support	8 SP	30	0	**7**
Upgraded Turret	6 SP	75	**15**	**3.5**

Game Rounds and Game Termination

Action Phase : At each round, after the deployment phase is completed, the game engine sets up Mobile units and Structures as players decide in the Deploy phase. The Action phase moves forward in steps and lasts until all Mobile units are either destroyed or reach the opponent's edge. When a mobile unit reches to one of the opponent's edges, it causes a breach and thus decrements the opponent health by 1.

Additional Allocations : Until one of the players reach to the HP of 0, or until the 100th round is reached, the players continue playing the game while receiving additional SP and MP to incentive deployment. Both players start each round by receiving an additional allocation of 5 Structure points and 5 Mobile points. Additionally, the game rewards players with 1 extra Mobile point for every 10 turns that have elapsed, increasing their capacity for strategic deployments as the game progresses—for instance, players receive 5 Mobile points from turns 0 through 9, 6 Mobile points from turns 10 through 19, and this increment continues in a similar pattern throughout the game.

Endgame : The game ends if a player reaches Health Point of 0, in that case the opponent will win. Additionally, the game will terminate in the 100th round given that neither of the players reached to HP of 0, in which case the player with the highest health will win. Finally, in the case that both players have the same health at the end of the 100th round the algorithm with the least computation time will be the winner.

Data Preparation

Web Scraper : Substantial groundwork was involved in building the infrastructural material that allowed for curation of data into a format suitable for machine learning. Terminal exposes an outward, player-facing replay file that is downloadable in a JSON-like format for each public terminal match. It is worth noting that the public matches follow a convenient URL schema and are thus primed for visitation from a web scraper. The web scraper used to find public matches specifically targeted those matches at indices near season kick offs. Heuristically, our team decided that this was a good approximation of determining dense swaths of matches that are likely to yield more advanced players. Our team ran a total of 30 instances of the web scraper over a course of 3 days to yield approximately 28,000 non-duplicate public matches concentrated around the season 7 Terminal kick off.

Figure 2: Schematic of data pipeline parsing phase

Extracting Desired State : The public replay files represent a log of the entire playable game, including animations related to the frame sequences. It was necessary to parse the state information we determined to be useful before using the data for model training. Our team went through two separate iteration cycles in regard to the state information we parsed from raw data, the second iteration being more comprehensive in that it incorporates more action responses and passes

more analytical state information for each (x,y) location on the game board. A further section will describe the motivation for this enlarged state designed in the second iteration. Reference Table 2 for differences between the state parsed in the first iteration cycle compared to the second iteration cycle. Here, state broadly refers to all game information, including the action response. *Game Board State* refers to the information stored for each cell on the board. For our *Action Response*, since players can play multiple actions in a single turn, we tally the number of times they play each action.

Table 2: Differences in Parsed State Between Iteration Cycles

Data Type	Parser Iteration	
	One	Two
Game Board State		
Attack	✓	✓
Current Health	✓	✓
Max Health	✓	✓
Range	✓	✓
Type	✗	✓
Owner	✗	✓
Breach	✗	✓
Cumulative Breach	✗	✓
Player State		
p1Health	✓	✓
p2Health	✓	✓
p1PreviousHealth	✓	✓
p2PreviousHealth	✓	✓
p1 Structure Points	✓	✓
p1 Mobile Points	✓	✓
p2 Structure Points	✓	✓
p2 Mobile Points	✓	✓
Round #	✓	✓
p1 Left Breach	✗	✓
p1 Right Breach	✗	✓
p1 Cumulative Left Breach	✗	✓
p1 Cumulative Right Breach	✗	✓
p2 Left Breach	✗	✓
p2 Right Breach	✗	✓
p2 Cumulative Left Breach	✗	✓
p2 Cumulative Right Breach	✗	✓
Action Response		
714 integers	✓	✗
1134 integers	✗	✓

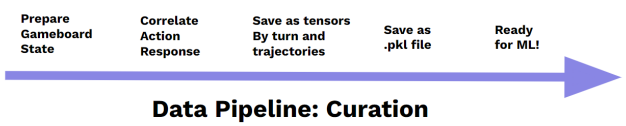

Figure 3: Schematic of data pipeline curation phase

Design Choices Among Parser Iterations : Refer to Table 2 for the following discussion of differences between the parsed state of the two iteration cycles. After evaluating performance in the first iteration cycle, our team made two key realizations: first, model awareness of which border cells are most breached can help in the placement of more efficient defense units. A simple terminal agent will likely not learn of key structural weak points in its defense; indeed, a smart opponent will then be able to exploit these weak points for a quick victory. Thus, for each (x,y) coordinate location, it becomes important to track both the breach status of the most recent turn as well as the cumulative history of breaches at that location. Note that as an invariant the breach data for any non-border cells is always 0.

The second realization was that advanced players focus on quality of structural units rather than quantity. Specifically, a simple agent places many low value defense units, inevitably congesting its own board space and hindering the viability of its mobile units, whereas advanced players upgrade key defense units and often remove unnecessary units in order to improve general percolation on their side of the game board. Thus, it also became necessary to facilitate upgrade and remove actions for any defense unit. Previously, the terminal agent performed a BFS to determine percolation status from border to the middle of the board before placing defense units so as to ensure a path for mobile units. Ability to remove units will significantly assist in the practicality of this BFS calculation as it allows the model more flexibility over turns and thereby allows for a degree of longer-term planning.

Data Sampling : Upon preparation of the data as outlined previously, the game board state, player state, and action response are processed into tensors and provided to the machine learning pipeline. It should be noted that batch sampling selects random indices from the dataset such that the game board state, player state, and action response for a given turn maintain a relationship while the turns in a given sample are not necessarily from the same game.

Training an Agent: Behavioral Cloning

As discussed in prior sections, there were 2 iteration cycles for data collection / extraction. For both cycles, we focused on training an agent via running Behavioral Cloning (BC) on the dataset of expert demonstration replay files we scraped from the competition website. We treated the winner of each match as the expert to imitate and remapped the points on the board so that the winners always played as Player 1 in the game (i.e. our agent learns that the expert only controls points with y-coordinate ≤ 13).

Architecture & Intuition For both iterations, we tested CNN-based and ResNet-based architectures for our policy network. The intuition behind this is: because we define our game board state in accordance with *Table 2* where we have

information at a cellular, per-point-on-the-board basis, this yields information in a 28x28xd matrix format (where d is the number of features per cell). Especially considering the game mechanics where the proximity of units has a bearing on nearby units, it intuitively makes sense to apply convolution layers to learn spatial information about the board and produce a latent vector encoding.

In addition to the game board state, for each round we have information at a macro level about the players' states (specifically regarding the players' healths, damages taken, points available for expenditure). This information is incompatible to be passed through 2D-convolutional layers. Thus, we first pass the game board state through the convolutional layers, obtain the output encoding, and then pass this encoding of the game board as well as the player state vector through linear layers to finally produce our action distribution.

We take our Action Response vector which contains the tallies of actions played in each round, turn it into a probability distribution by calculating the proportion each action was played per round, and then compute the Categorical Cross-Entropy Loss between our model's output and the true frequency the actions were played with.

There are a total of 504,000 samples in our dataset, split 95% for training, 5% for validation so as to maximize the data available for our model to learn.

A diagram of the architecture can be found below.

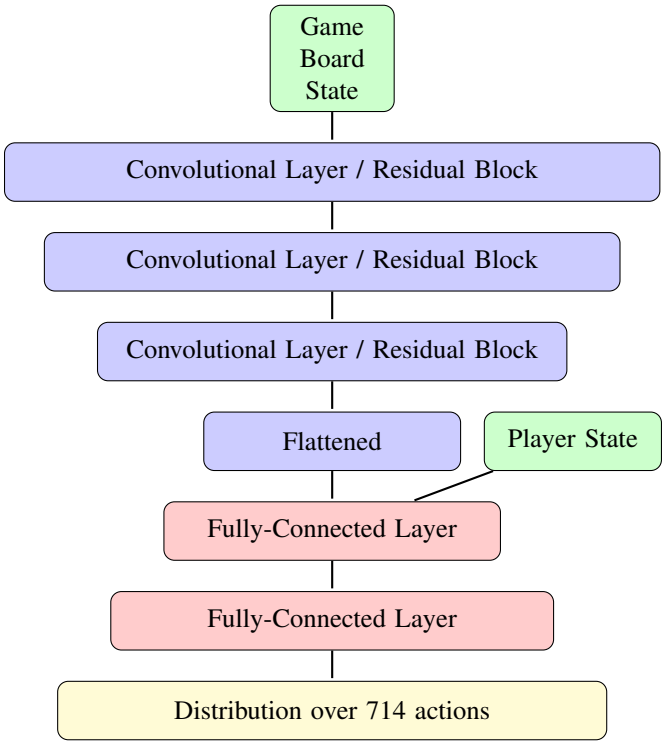

Figure 4: Policy Network Architecture

Inputs & Outputs :

Iteration 1: For our first iteration of the data collection / models, our game board state is a 28x28x4 matrix containing attack, current health, max health, and range for every cell on our game board (as shown in *Table 2*). Our Player State is a 9-dimensional vector. Our output is a distribution over 714 playable actions (only placing structure and mobile units).

Iteration 2: Game board state is a 28x28x8 matrix. Our Player State is a 17-dimensional vector (now containing breach information). Our output is a distribution over 1134 playable actions (placing, removing, upgrading structure and mobile units).

Model Parameters :

CNN: Every convolutional layer had padding to maintain the width and height of the inputs and was followed by Average pooling over 2x2 patches to then downsize the width and height, each by half. Every convolutional layer also had a kernel size of 5x5, stride of 1. The final output was of shape 3x3x32.

ResNet Details: There were 3 ResNet blocks. Each block consisted of 2 Conv layers followed by Batch Norms. The first ResNet block went from an input of depth 8 to 8 feature maps, the second block went from 8 to 16, and the third block went from 16 to 32 feature maps. The final output was of shape 2x2x32.

Fully-Connected Layers Details: The input to the first fully-connected layer was either the flattened 3x3x32 or 2x2x32 (depending on whether the CNN or ResNet layers were used) followed by either a 9 or 17-dimensional vector (for iteration 1 and iteration 2, respectively). The following layers then have 512 and 1024 neurons, with the final layer outputting a distribution over either 714 actions (for iteration 1) or 1134 actions (for iteration 2).

Deploying the Agent

After training a policy network via Behavioral Cloning, we are ready to deploy it and test our model against other algorithms and strategies in real-time. Below is a general procedure for how we do so:

1. Save model parameters in a '.pth' file after training. Load the model in the Python strategy file we submit to the Terminal competition leaderboard.
2. Reconfigure the Terminal starter kit code and use their API to extract/process the state information during each round to get the desired features and input format for our model.
3. Run inference on our model and get the probability distribution over the actions.
4. Make moves heuristically:
   (a) While we have more structural points: mask out the actions corresponding to placing mobile units / placing units on already-occupied cells. Then sample from the probability distribution over remaining, playable actions and build structural units.

(b) If the round number is even, while we have more mobile units: mask out the actions corresponding to placing structural units / placing units on structure-occupied cells**. Then repeatedly sample and place mobile units. *We only place mobile units every other round is so we can conserve mobile points, then deploy a large group of mobile units which have a better chance of breaching the opponent's defenses.*

** We additionally, need to determine whether a mobile unit placed on our border will be boxed in by structural units (i.e. can it reach the opponent's side or get trapped?). We check this with a Breadth-first search from the cells on the opponent's border with us, and traverse to see which cells on our side of the board we can reach. We then mask out the actions in our probability distribution that correspond to placing a mobile unit on boxed-in cells as those mobile units would be trapped by structures and serve no purpose.

In this manner, we can test our model against other players' strategies on the leaderboard, as well as the 5 bots of varying difficulty on the Terminal website.

Results / Experiments

The graph below shows the Categorical Cross-Entropy losses (training and validation) of a CNN-based architecture and ResNet-based architecture after iteration 2 of data collection.

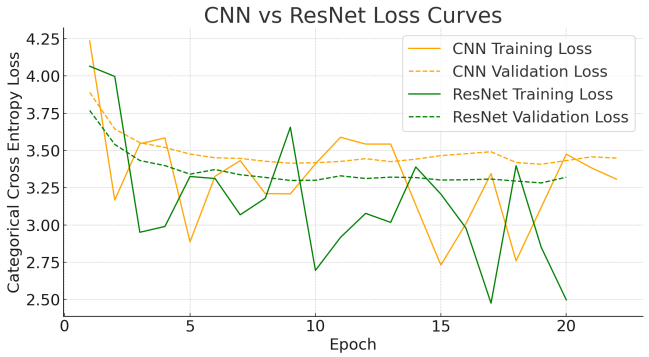

Figure 5: Training and Validation Loss Curves for Iteration 2 of CNN and ResNet architectures (as detailed in *Training an Agent* section).

As can be observed from *Figure 5*, training loss repeatedly increases and decreases across epochs for both models while the validation loss decreases and then tapers off with diminishing returns and smaller fluctuations. Additionally, it can be seen that the ResNet model achieved lower validation losses with one of its low points being around 3.299 at Epoch 9.

At Epoch 9, the CNN-based model also achieved a relatively low validation loss, prior to it having higher validation losses for the next 7 epochs. Thus, the model's parameters at this epoch were chosen for later testing on the Terminal website. Similarly, Epoch 9 was chosen for the ResNet-based model as the model had not yet overfit on the training set. Epoch 19 was also chosen to test in practice as it yielded relatively low validation and test losses.

A nice aspect of training an agent to play the Terminal tower-defense game is the ease of testing the model in the real-world against other strategies. We took several different trained models, with different hyperparameter configurations and different epoch lengths of training, and pitted them against each of Terminal's 3 bots 10 times.

The results are shown in the table below (bots increase in difficulty left to right):

	Bot Levels		
Model	**Easy**	**Medium**	**Hard**
CNN Iter 1	9 / 16.3 / 1	7 / 27 / 10	2 / 31.5 / 22.1
ResNet Iter 1	10 / 28 / 0	7 / 30.7 / 10	7 / 30 / 10.3

Table 3: Each model from Iteration 1 played 10 games against each of the Bots. The first number in each cell denotes the number of wins out of 10 rounds. The second and third numbers are the average absolute health difference at the end of rounds the model won and lost, respectively.

As can be seen, from Iteration 1, the ResNet model seemed to perform better in practice, generally. It won just as many matches, if not more matches, at each of the difficulty levels than the CNN model. Additionally, with the exception of the hard bot, when it was played against the easy and medium bots, for the rounds it won, it had a larger health difference over its opponent (28 and 30.7). When the ResNet model lost rounds to the bots, it lost by a smaller margin than did the CNN model (losing by 0, 10, and 10.3 health on average).

Thus, as observed from *Figure 5* and *Table 3*, it appears the ResNet model architecture yields the best results, both in terms of loss curves (Iteration 2) and in-real-world play (Iteration 1).

Limitations and Future Work

Heuristic Choices : The state space of terminal is immensely large; certain heuristic choices must be made when determining what state may be valuable to a model during training. Moreover, the choices our team made adapted throughout the process of constructing our Terminal agent, as can be seen with the differences in state between parser iterations (see Table 2).

Additionally, a large heuristic choice centers on whether or not action responses are separated between Structure units and Mobile units. It should be noted that both of these unit types draw from different resource pools, in which sense it would be logical to separate policy networks into two more specialized networks. However, such a choice may eliminate the possibility for coordination between structure and mobile units, which is a key element of more advanced strategies. Moreover, the objective of learning coordination between unit types was precisely what motivated our second parser iteration cycle. Our team decided upon a single policy network for these reasons, but it should be noted that future approaches

may benefit from separate policy networks for attack and defense.

Data Quality : Behavioral cloning methods function most optimally when an agent learns a policy based on expert data. However, there is no specific indication that our data collected at large can be deemed "expert data". While our team roughly selected data from intervals corresponding to season kick-offs, it is certainly true that our model is learning from both the winners and the losers of these matches. Future iterations of data collection may involve a more involved web scraper that is able to scraper for specific players found on public leader boards (top 500 worldwide - which would now include our matches as well!).

Self-Play Environment : A crucial, exciting area of future development in the field of designing a Terminal agent is ability to perform self-play to avoid the large data overhead needed to perform behavioral cloning. Moreover, model-based reinforcement learning methods may yield enticing results for a Terminal agent given the highly deterministic nature of the game. Our team has designed reward functions that heuristically combine several features of interest to assign rewards to actions: maintenance of the agent's health, decrease to opponent's health, building contiguous segments of structural units (calculated via Depth-First-Search + Manhattan Distance of connected components), and number of structural and mobile points left. While our team has been developing a self-play environment for Terminal that would open doors to many more avenues of research for the open-source community, this project is still in development.

Conclusion

We have successfully demonstrated how behavioral cloning techniques can be applied to the Terminal tower-defense competition and outperform human-crafted, rule-based algorithms and strategies. Whereas in the past, players would write hundreds or thousands of lines of code to conditionally carry out actions based on the game state, we have deployed a policy network that is able to win a majority of games against Terminal's hardest Ironclad bot. Our utilization of behavioral cloning places our agent within the top 500 of over 154,000 (top 0.35%) competing algorithms and demonstrates the promise of Imitation Learning and Reinforcement Learning techniques for adapting to a strategy-based, tower-defense game like Terminal.

Our GitHub repository with data collection, model training, and deployment code can be found here.

Acknowledgments

We would like to acknowledge and thank Professor Mengdi Wang and Professor Benjamin Eysenbach for their kind feedback and guidance on this project.

References

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*. Retrieved from https://arxiv.org/abs/1512.03385

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998, November). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324. doi: 10.1109/5.726791

One, C. (n.d.). *C1gamesstarterkit.* https://github.com/correlation-one/C1GamesStarterKit. ([Software])