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

Yagiz Devre [hd0216@princeton.edu] [1] Ishaan Javali [ij9461@princeton.edu] [1] Shivam Kak [sk3686@princeton.edu][1]

[1]Princeton University

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 & Game Mechanics

Correlation One's Terminal is a Tower Defense game set in a diamond-shaped arena, where players use mobile units and structures to balance offense and defense. Players utilize two resources: mobile points (MP) for attack units and structure points (SP) for defenses, aiming to reduce the opponent's health to zero. The game combines planning, forecasting, and real-time decision-making.

This paper explores using Reinforcement Learning (RL) to master Terminal's strategic challenges, surpassing human gameplay capabilities. The game's complexity requires both strategic foresight and adaptability, presenting a unique challenge for traditional game-playing algorithms.

Until now, players have relied on rigid, rule-based methods which lack flexibility against novel strategies.

RL offers a viable alternative, allowing dynamic strategy formation and adaptability. Terminal's gameplay requires simultaneous actions each round, demanding predictive and strategic adjustments. The game's large state and action spaces, along with the indirect nature of its rewards, make RL well-suited to navigate and optimize strategies effectively.

This project aims to develop an RL algorithm that excels in Terminal and provides insights applicable to real-world strategic decision-making environments, advancing AI's capabilities in dynamic and competitive settings.

State and Action Space

Game Description: Terminal is a Tower Defense game where two players engage in simultaneous turns on a diamond-shaped grid. Each player starts the game with 40 structure points (SP), 5 mobile points (MP), and 30 Health Points (HP). The game challenges players to strategically deploy units and structures for offense and defense within a competitive setting.

Game Board and Setup: The board, as in Figure 1, is a 28x28 diamond grid divided into two halves, each controlled by one player. The game begins with no units on the board, setting the stage for strategic deployments.

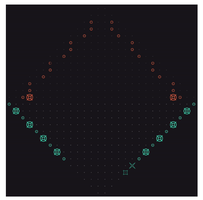

Figure 1. Game Board Layout

Unit Deployment and Actions: Players have the option to deploy three types of structure units and three types of mobile units:

- Structure Units: Wall, Support, Turret.
  - Wall: Prevents opponent Mobile Units.
  - Support: Provides shielding to friendly units.
  - Turret: Deals damage to an enemy Mobile units.
- Mobile Units: Scout, Demolisher, Interceptor.
  - Scout: Fast-moving mobile unit with light damage.
  - Demolisher: Slow-moving unit with high damage.
  - Interceptor: Very-slow-moving unit that deals damage to enemy Mobile units.

Structure units can be upgraded or removed to adapt strategies, allowing players to reclaim a portion of spent SP. Mobile units are placed on the edge points of each player's side, with possibilities for stacking multiple units at a single location for strategic advantages. All the attributes of the units 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 tiles
Demolisher	3 MP	5	8	4.5 tiles
Interceptor	1 MP	40	20	4.5 tiles
Defense Units				
Wall	1 SP	60	0	0 tiles
Support	4 SP	30	0	3.5 tiles
Turret	2 SP	75	5	2.5 tiles
Upgraded Defenses				
Upgraded Wall	2 SP	120	0	0 tiles
Upgraded Support	8 SP	30	0	7 tiles
Upgraded Turret	6 SP	75	15	3.5 tiles

:Game Dynamics and Phases: Each game round includes a deployment phase followed by an action phase where mobile units advance towards the opponent's edge to attempt a breach and reduce HP. The game provides additional SP and MP allocations each round to enhance strategic options.

Endgame Conditions: The game concludes when a player's HP reaches zero or at the 100th round. The player with the highest remaining HP wins. If HP is tied after 100 rounds, the winner is determined by the least computation time used.

Data Preparation

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.

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. Second, advanced players focus on few, upgraded units. Thus, for each (x,y) coordinate location, it becomes important to track both the breach status of the most recent turn and cumulative history of breaches at that location. It also became necessary to facilitate upgrade and remove actions for any defense unit, which complements our existing BFS detection.

Figure 2. Data Pipeline Parsing Phase Schematic

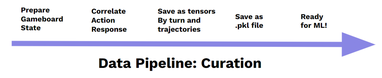

Figure 3. Data Pipeline Curation Phase Schematic

Agent Deployment

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 ours model.
3. Run inference on our model and get the probability distribution over the actions.
4. Make moves heuristically:
   1. While SP remaining: mask out the actions corresponding to placing mobile units / already-occupied cells. Then sample from the probability distribution over remaining, playable actions.
   2. 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.

Training an Agent: Behavioral Cloning

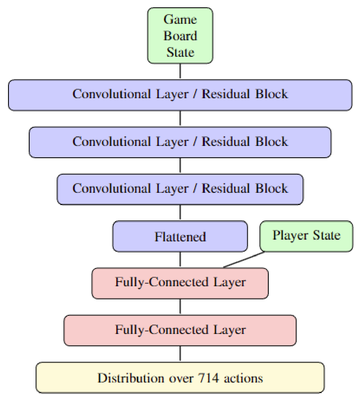

Figure 4. Policy Network Architecture

:

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. 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).

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).

Results & Experiments

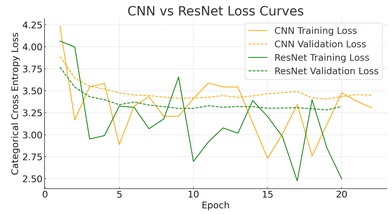

Figure 6. Training and Validation Loss Curves for Iteration 2 of CNN and ResNet architectures (as detailed in Training an Agent section).

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.

	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 2. 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 & Future Work

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.

Behavioral cloning methods function most optimally when an agent learns a policy based on 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!).

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. 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.

References

[1] 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

[2] 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

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

Acknowledgements

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