Final Project
for the course 67842 Introduction to Artificial Intelligence

Jenga AI Solver: Agents for Strategic Block Removal

By Alon Biner, Mariia Makarenko and Yonatan Vologdin

Hebrew University of Jerusalem
2024

Introduction

In our project we are solving the problem of developing an AI agent that performs well in the game of Jenga. For those not familiar with the rules, the game starts with 54 wooden rectangular blocks (although there are variations of the game with less blocks) stacked in a tower. At each level of the tower there are three blocks placed side by side along their longer side, and the levels alternate direction relative to each other. For example, if in the downmost level the blocks are oriented horizontally by their longer sides, in the next level the blocks should be oriented vertically. The players make moves one after the other, so that in each move a player removes one block from any layer, except the top three, and places it on top of the tower. The player who causes any of the blocks to fall loses the game.

The more blocks are removed from the tower, especially from the lower levels and from the sides, the more unstable the tower becomes and the more likely it is to fall. To succeed in the game, a player needs to make a move that is cautious enough for the tower not to fall in their move, but also risky enough for the tower to become significantly more unstable in the adversary's move. Due to this tradeoff, there is no trivial strategy the player can adhere to in order to win, and this makes the problem interesting.

As can be inferred already from the description of the game, Jenga is a physical game. In the real-world Jenga game, noise is introduced to the players' strategies due to imperfect tower stacking, manufacturing tolerances, and friction between blocks. That is, no matter how good their strategies are, the tower can fall if they accidentally touch other blocks when pulling out the block they want to. Because of this, normally players can "probe" the blocks to find the ones that are loose enough to be safely pulled out. Therefore, to create a solver for the real-world Jenga, we would need to use robotics. Since this is not a project in robotics, we decided to simulate the game in its simplified form.

For simulation of the Jenga tower, we turned to Unity and its physics engine. It allowed us to control the parameters of static and dynamic friction, as well as gravity. It also allowed us to know the positions of all the blocks at any given time. First and foremost, it allowed us to isolate the strategic component of the Jenga game by eliminating the noise when pulling out blocks. That is, we were able to implement the process of pulling out of the blocks as if they are pulled without any friction with the neighboring blocks. Furthermore, it allowed our agents to get understanding of the current stability of the tower based on the blocks' angles and, for example, make more cautious moves when the tower is less stable.

Now, before you embark on a journey of exploring our project, we would like to mention two things:

- We answered some of the questions that aim to "expand" the project (e.g. how assumptions made may affect the solution to the original problem, or if we can generalize our model to other problems) directly in the summary section.
- We used ChatGPT for the tedious task of writing code documentation.

Previous Work

In our search for previous work that tried to make an automated solver for Jenga, we managed to find only one paper on the matter, which is a project by a team of researchers from MIT (N. Fazeli, M. Oller, J. Wu, B. Tenenbaum, and A. Rodriguez) published in the Science Robotics journal, with a short video available on Youtube about it. Their project aimed to create a solver for Jenga in the real-world setting and described creating a robot that is capable of seeing and touching. One of the downsides of this approach is that physical learning is very costly. Our project, on the contrary, only aims to solve Jenga in a simulation, although we believe that with parameters tweaking in Unity we can achieve high degree of realism.

Similarly to the "probing" technique used by the human players, the robot uses tactile feedback to locate loose blocks that are safe to remove. It selects a block at random, tries to push it for some time, and if the push doesn't make the block stick out, it retracts and goes to the next random block. As well as us, the researchers assumed that the blocks are rigid and don't change form when force is applied, are absolutely equal, and there are no external forces like the wind and vibration.

During a short training period that the researchers call exploration phase, the robot learns the underlying physics model of the game: it categorizes the blocks into groups and adjusts the force and its direction according to the current block's group. After the exploration phase the robot's performance is measured. Essentially, the robot learns to play Jenga without the opponent, and so its main goal is to prolong the game by making the safest move. On the other hand, the solver in our project has an opponent, and so its objective is to try to make the riskiest move that doesn't fall the tower in order to win.

The researchers used Markov Chain Monte Carlo (MCMC) Sampling with Hamiltonian dynamics to approximate the distribution over states and abstractions after receiving noisy observations. While the regular Monte Carlo draws independent samples, MCMC draws samples so that every sample depends on the previous one. Hamiltonian dynamics means that the sampling is done according to some clues. The researchers also used Deep Residual Learning, which learns the difference between the desired output and the input. In our project we either tried or actually used the simpler counterparts of these algorithms, Monte Carlo Sampling and Deep Learning.

The robot's performance was evaluated by how many moves it could make in a randomly generated tower until the game ended. In our project we made a similar metric of the average game duration against the adversary. As baselines, the researchers used a Feed-Forward NN model and PPO implementation from the OpenAI Gym. In our project we also used a Feed-Forward NN model that was trained with fewer samples, as well as an adversary with random strategy.

Methodology

The codebase of the project is divided into two parts: the Unity implementation, and the Python code for all other aspects of the project and uses the Unity build created from the first part. The first part can be found here, and the second part with the instructions on how to run every part of the project can be found here.

When running the project, Python code and Unity build run together at all times and communicate over TCP. Unity physical engine is responsible for handling block collision simulations, as well as detecting whether the game ended using colliders around the tower. Unity is capable of resetting the tower, removing a block, detecting any of the blocks falling, and calculating the blocks angles. The agents in Python send commands to Unity to perform the removal of the block they need and get feedback on the state of the tower after the action was performed.

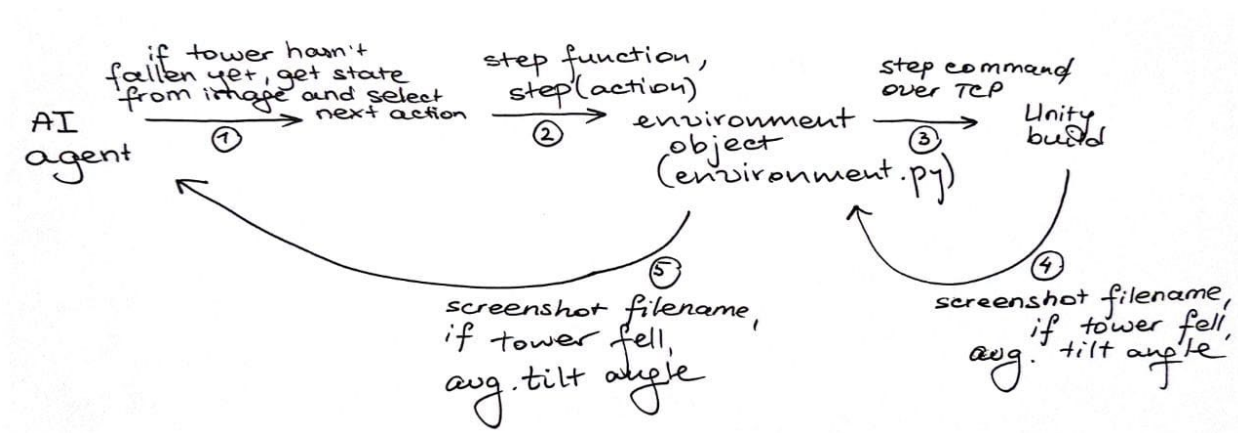

Figure 1: High-level communication process

We assume in the project that the blocks are rigid, are absolutely equal, are assembled in the tower perfectly, and there are no external factors like the wind and vibration. We also assume that Unity models everything correctly according to the environmental parameters passed, and that these parameters don't give advantage to any specific strategy of the player. Other assumptions are mentioned later in this section.

Since blocks can tilt and shift, representing a game state using a binary string that has zeros and ones depending on whether blocks are present or not wouldn't be enough. And since blocks that are arranged parallelly hide those behind them, observing only one side of the tower wouldn't suffice either. Therefore, we use a 128×64 "spread" of the tower – combined screenshots from two adjacent sides, – which provides enough spatial context.

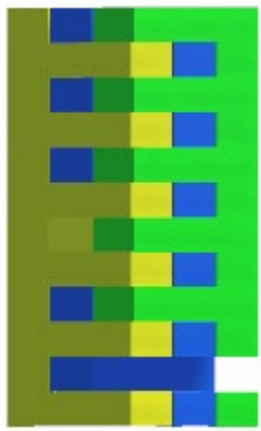

Figure 2: Example of a spread of the tower

The spread is converted to greyscale and normalized to $[-1,1]$. So, the state space of the problem is as follows:

$$S = \left\{ s \in \mathbb{R}^{1\times128\times64} \mid -1 \leq s_{i,j} \leq 1 \right\}$$

Using image files had its own flaws. We had to introduce delays as sometimes the images created by Unity were not registered by the operation system right upon the agent's request. Also, it made most of the agents highly sensitive to their input.

To manage the large branching factor in Jenga, and to make the learning process more feasible for the agents, we reduced the tower height from 18 to 12 levels. Additionally, we assumed that when removing a block, it doesn't touch any of its neighbors. And finally, we removed the need to place an extracted block on top of the tower.

We numbered each level starting from zero, with zero being the top-most level. Also, for discerning between blocks at the same level from different points of view, we colored blocks at each level in Unity in three colors and assigned them numbers: yellow (0), blue (1) and green (2). Each action is represented by a tuple of the number of the level and the color of the block to remove, and so the action space is as follows:

$$A = \left\{ (l, c) \mid l \in \{0,1,\dots,L-1\}, c \in \{0,1,2\} \right\}$$

We can notice that in this problem Jenga is a stochastic game in the sense that the outcomes of the same action can differ between games due to the different stabilities of the tower, as well as between simulations in the same game due to non-deterministic physics simulation by Unity.

We defined the reward function the same way for all the agents. We assume that the lower level a block is taken from, the more unstable this makes the tower. This is because we assume that in the simulation all blocks lie firmly on each other, since aren't any manufacturing errors or inaccuracy in assembling the tower. Because we want the agent to make the tower more unstable for the adversary, we give a bonus for the removed block's level. However, we want the agent to not make the tower too unstable for it to fall, and because the consequences of the same move differ drastically between the games, we give a proportional penalty or reward for making the average tilt of the blocks bigger or smaller after the move. The information on the tilt is calculated by the Unity build after block removal as the average tilt of all the blocks, where for every block the tilt is the maximum between the axes.

Initially we thought that Monte Carlo Tree Search (MCTS) would be a good fit for the project. However, with the branching factor still being big after tower reduction, and with games ending relatively quickly, expanding beyond a single level in the search tree would be not only computationally costly, but also unnecessary. With only one level being expanded, it was no longer MCTS, but rather a greedy algorithm which we called Greedy Simulation-Based Action Search (GSBAS).

Every time before choosing a move, the GSBAS agent randomly chooses ten actions out of possible actions and simulates them. We chose to simulate only ten because simulations are slow, and we can count that even within these ten there will be good actions. GSBAS then takes the action that achieved the highest reward in the simulation. The problem in this approach is that the reward function is significantly smaller if the tower fell. So, the agent knows after which actions the tower almost certainly falls in the current game, and this gives the agent unfair advantage.

The other two agents we implemented are based on reinforcement learning and use the Deep Q-Network (DQN) approach to estimate Q-values of actions. They both are called hierarchical agents because they use two separate Neural Networks for the estimation: one for selecting the level in which a block is removed, and the other to select the color of the block. Here we are assuming that blocks in the same level and blocks of the same color can be grouped together. The regular Hierarchical DQN agent assumes the environment is deterministic and updates the Q-values based on the best action, no matter if this action was actually taken (off-policy RL). Meanwhile, SARSA assumes the environment is partially

stochastic and updates the Q-values based on the action it actually takes next (on-policy RL).

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$$

Since the next action the SARSA agent chooses can be an exploration action, this leads to more conservative learning and less overestimations.

Results

We created several baseline strategies for adversaries to evaluate the performance of our models against:

- Random Strategy – the adversary chooses a random block in their move.
- Optimistic Strategy – the adversary thinks the tower is more stable than it actually is and chooses a random block from the same level or lower than the agent did in its last move.
- Pessimistic Strategy – the adversary thinks the tower is less stable than it actually is and chooses a random block from the same level or higher than the agent did.

And we used the following evaluation metrics for all the models:

- Win Rate – the percentage of games the agent wins, including games that ended in a tie.
- Average Moves Until Collapse – the average number of moves in one game made by both the agent and the adversary before the game ended.

When plotting the results according to both evaluation metrics for the models that learn, we plotted how the results change depending on the number of episodes (i.e. games) played against itself from the previous training iteration or against the Random Strategy adversary. We will call this additional performance metric Learning Stability.

To ensure clarity of the results, we implemented several measures:

- For the Hierarchical DQN Agent and for the Hierarchical SARSA Agent, the evaluation phase was separate from the training phase and the agents didn't learn from the moves they made when being evaluated.
- During the evaluation phase exploration was disabled, ensuring that the agents act solely based on the learned policies rather than make random moves.
- The order of turns was randomized, so that no player gets consistent advantage of making the first move.

- To ensure proper timing in communication between Python and Unity so that all the state after the move that the agent gets is true, operational delays were added.
- The evaluation phase lasted long enough for the results to be consistent.

First, we will discuss the performance of the agents under the win rate metric. The following plot depicts the win rate of the GSBAS agent after playing 15 games against adversaries with baseline strategies we discussed earlier. The win rate against the Random Strategy adversary suggests that it performs at least as good. However, its win rate against the Optimistic and the Pessimistic strategy adversaries is low.

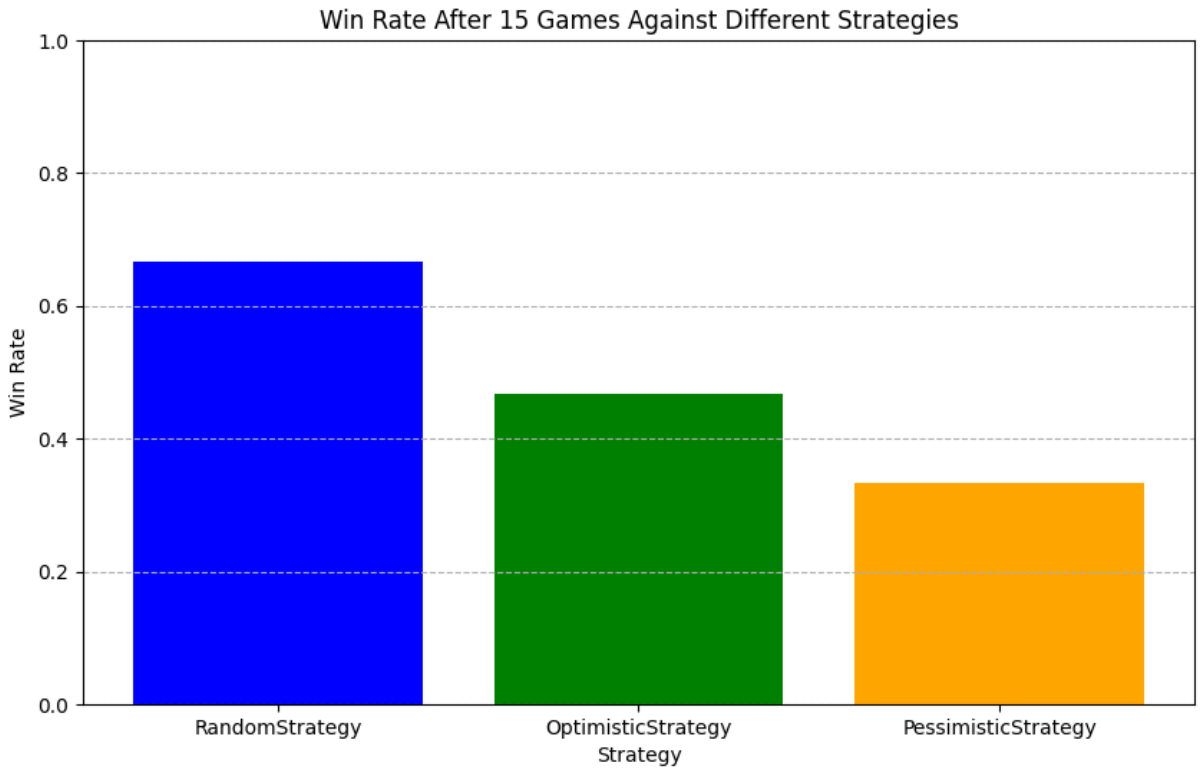

Figure 3: Win rate of GSBAS against different strategies after playing 15 games

This phenomenon can be explained by the fact that the GSBAS agent maximizes the reward function in real-time in the current game. By definition of the reward function, the agent prioritizes moves that remove lower blocks but do not make the blocks tilt significantly, which limits its ability to destabilize the tower for the opponent. As a result, GSBAS tends to be overly cautious (we can notice in Unity that it usually starts by removing all the middle blocks, which are the "safest" option) compared to other agents that maximize the reward function overall. Since a move that performed good overall is not necessarily the best move for the particular game, these other agents introduce an element of chance, often leading to bolder, riskier moves.

Another disadvantage of the GSBAS agent is its speed. We weren't able to parallelize the simulations the agent runs to determine the next move due to the enormous amount of resources a sole instance of Unity consumes. Because the agent performs the simulations sequentially and every simulation takes a lot of time, the decision-making process of the GSBAS agent is much slower than the one of its counterparts.

In the following plot you can see the win rate of the Hierarchical DQN agent after playing 20 games against adversaries in each evaluation phase, as a function of total number of training episodes against itself from the previous training iteration.

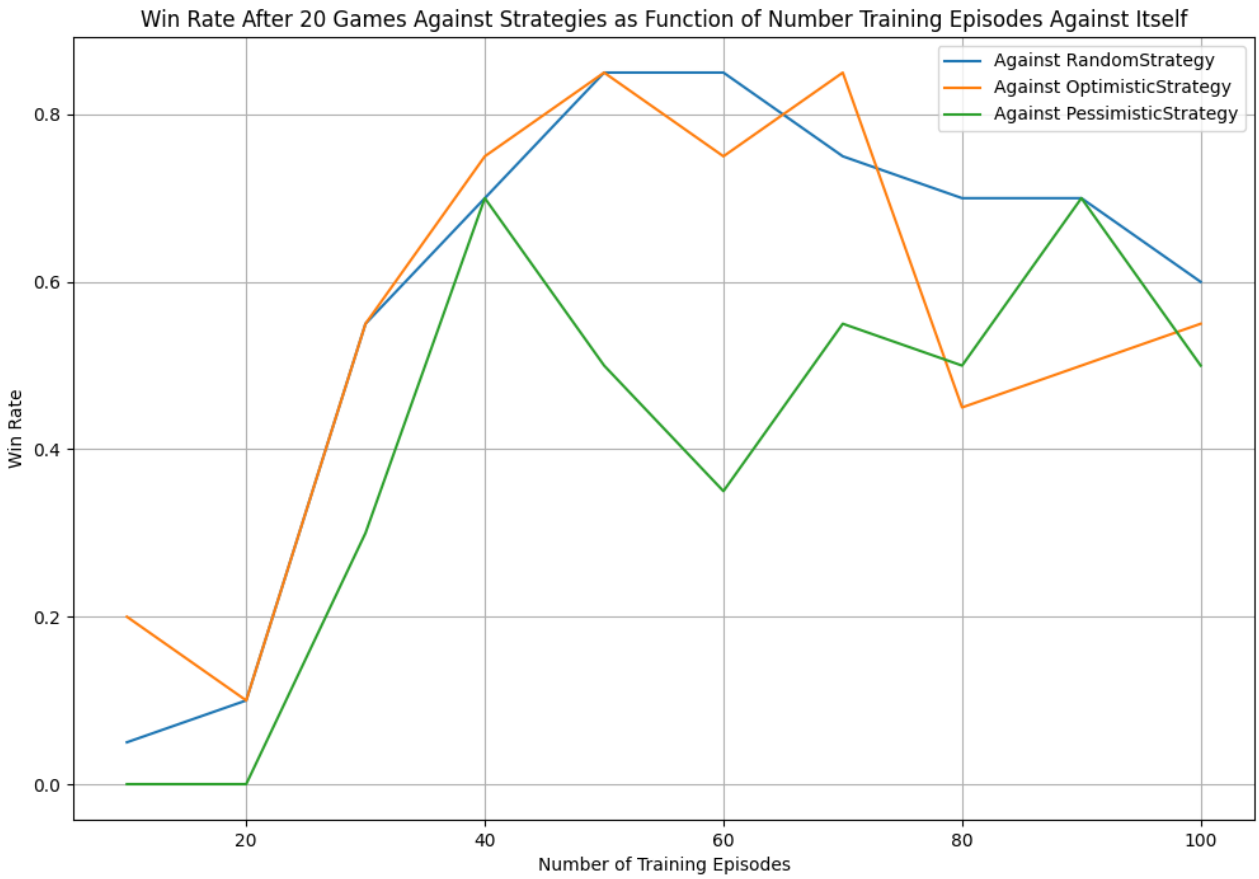

Figure 4: Win rate of Hierarchical DQN against different strategies after playing 20 games as function of total number of training episodes against itself

The low win rate at the start suggests that the random movements that the agent explored early on are too risky. After the agent adjusts the Q-values of these moves and explores safer alternatives, its performance improves dramatically. However, right after the fast improvement we see a dip in the win rate, in particular against the Pessimistic Strategy. This indicates the agent may be overfitting, choosing overly risky moves. While Random and Optimistic Strategies may fall for these traps, the Pessimistic Strategy is too cautious, causing the agent to collapse the tower itself in the subsequent move. Fortunately, by the

100th episode, the agent balances its risk and develops a more universally effective policy across strategies.

To prevent overfitting, we established a threshold of minimum desirable number of moves till the tower collapses. If the game ends before this threshold, it indicates that the moves that the agent makes are too risky and we increase the exploration rate (that is, we increase the epsilon in the epsilon-greedy policy) forcing the agent to try out random moves and eventually adopt a more cautious strategy.

Following is the plot of the win rate of the Hierarchical SARSA agent in the same setting as in the previous plot.

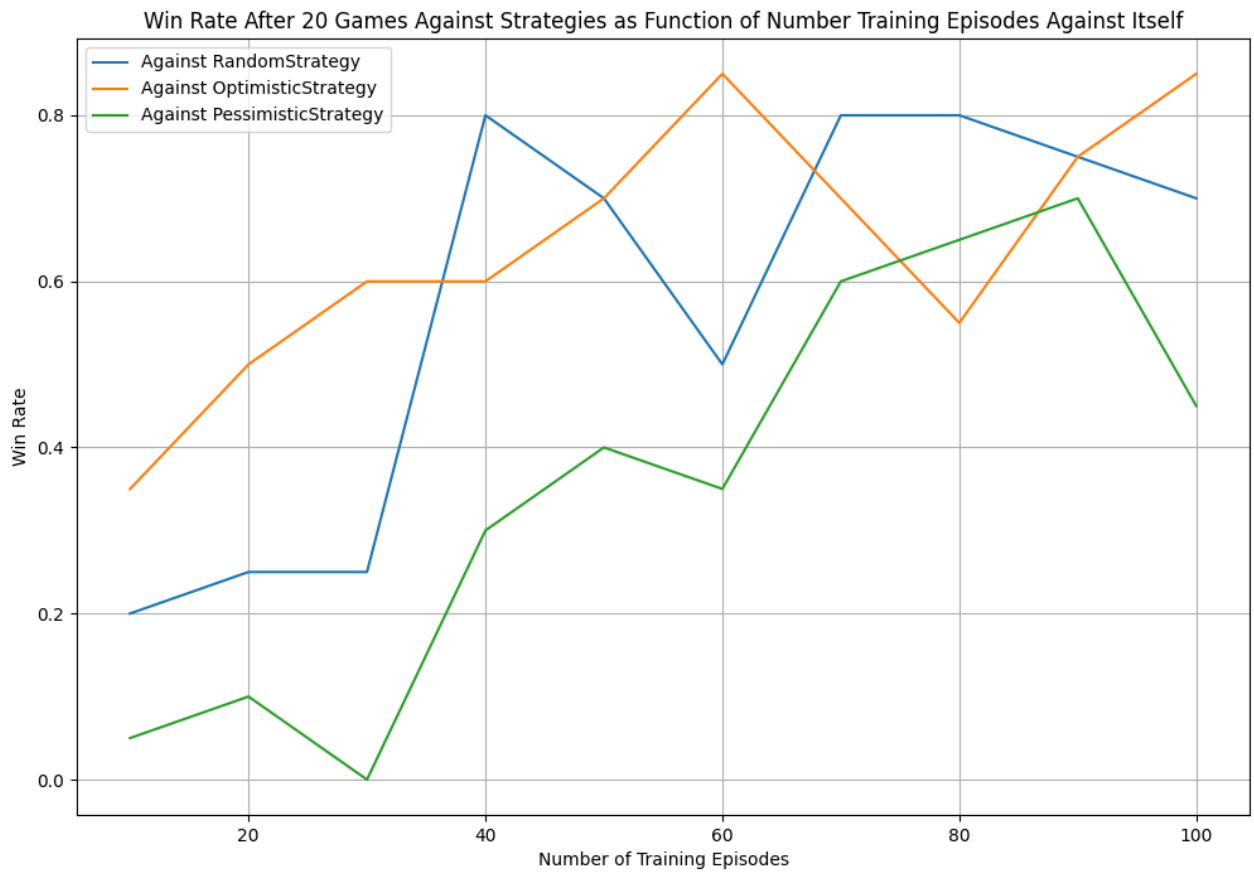

Figure 5: Win rate of Hierarchical SARSA against different strategies after playing 20 games as function of total number of training episodes against itself

The plot demonstrates a slower yet more stable improvement in SARSA compared to the regular Hierarchical DQN agent, suggesting that SARSA is more conservative and focuses on safer strategies. Indeed, since SARSA updates the Q-values based on the actions the agent actually takes, and not the hypothetical optimal ones like in the regular DQN, the resulting

Q-values are more realistic. On the other hand, the regular DQN agent tends to overestimate the Q-values it hasn't tried much, and usually those are actions that are too risky.

In the end SARSA achieves higher resulting win rate than the regular DQN agent, with particularly higher win rate against Optimistic Strategy. This is due to the fact that it makes more cautious moves, and so aggressive strategies like Optimistic fall into their own trap. However, it is on par with more cautious strategies like Pessimistic, since it is unable to destabilize the tower enough for them.

Overall, the agents struggled the most with the Pessimistic Strategy because it makes very conservative moves. But also, the strategy is defined in the way that the more conservative the agent itself is, the more conservative the Pessimistic Strategy becomes. Therefore, it is especially harder for more conservative agents to beat it.

And now we would like to point out some other observations on the learning process of the agents that their win rate highlighted. For example, that the agent performs better when trained against itself from the previous iteration rather than Random Strategy, as you can see in the following plot.

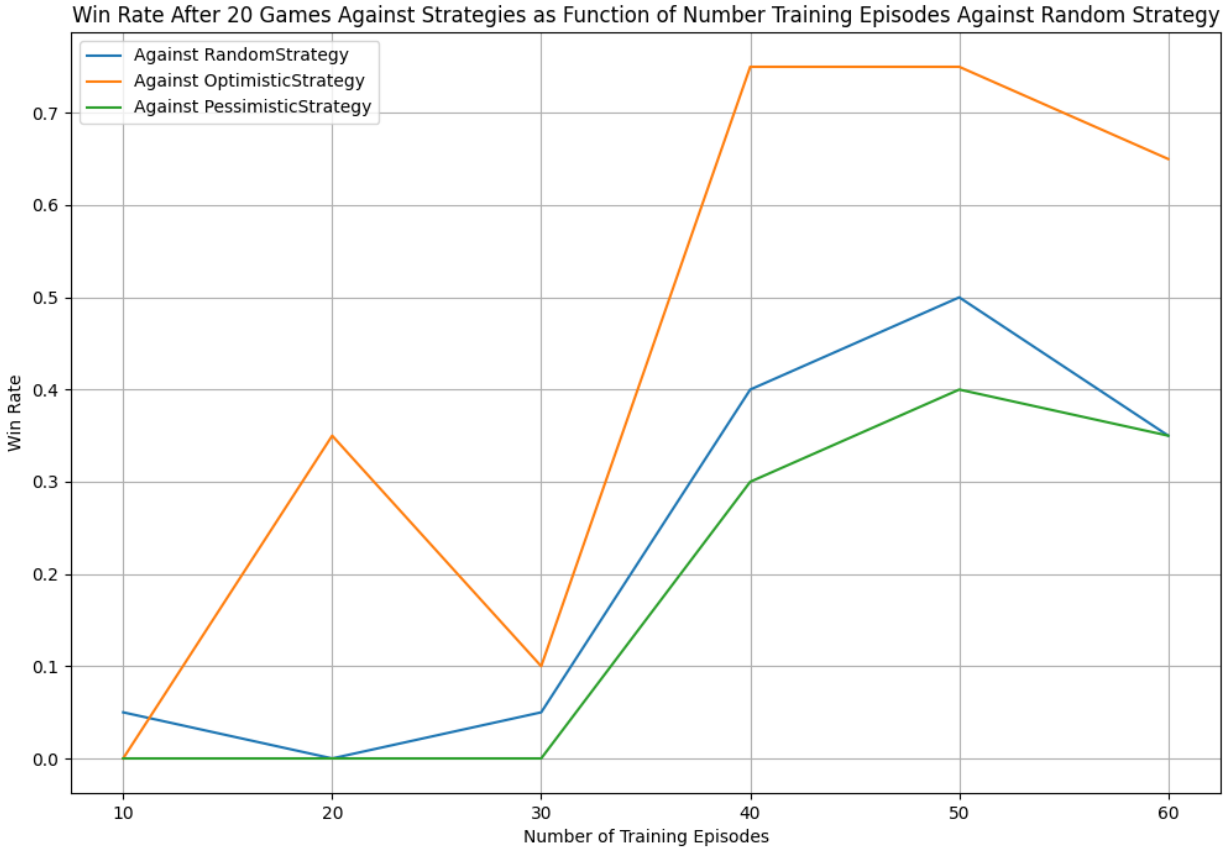

Figure 4: Win rate of Hierarchical DQN against different strategies after playing 20 games as function of total number of training episodes against Random Strategy

Although the increase in the win rate is smoother compared to training against itself, the win rate against Random and Pessimistic strategies is by far lower. This is due to the fact that when the Hierarchical DQN agent trains against itself from the previous iteration, with every new iteration of the evaluation phase the opponent improves. However, the Random Strategy opponent remains the same and doesn't introduce enough challenge for the agent to learn to overcome the Random and the Pessimistic strategies. Regarding the Optimistic Strategy, we can speculate that the win rate is high because the agent makes risky moves, and the riskier the agent's moves are, the riskier the moves of the Optimistic Strategy are to the point of making the tower fall.

The same can be said about the win rate of the Hierarchical SARSA agent when trained against Random Strategy, although the win rate is a bit higher for all the strategies.

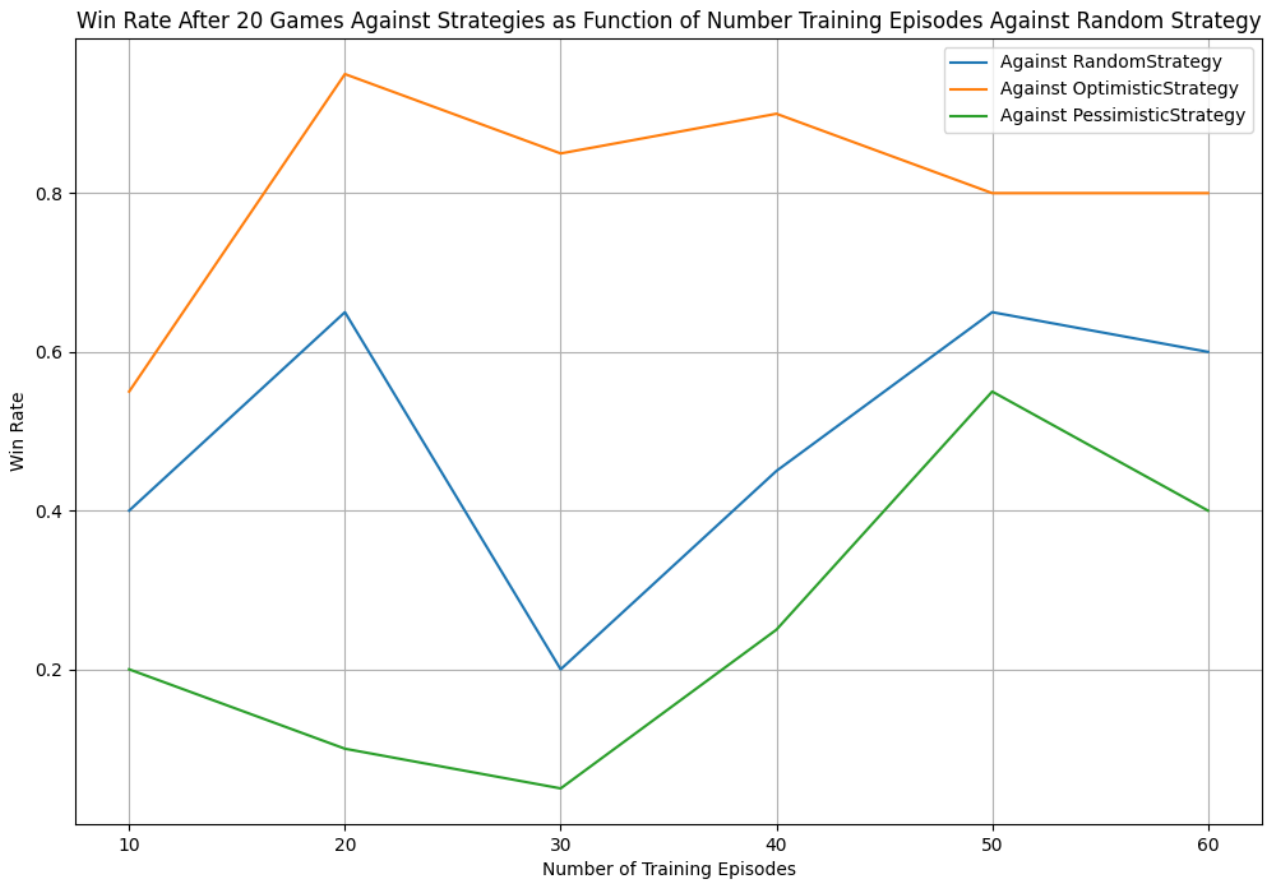

Figure 7: Win rate of Hierarchical SARSA against different strategies after playing 20 games as function of total number of training episodes against Random Strategy

Another observation is the agents perform better when trained with a "diverse" reward function. Initially, our reward function was comprised of reward for the level of the block being taken out and penalty for making the tower fall. However, as seen in the following plot,

this approach led to lack of learning, likely because the reward function assigned similar values to many different actions, making it harder for the agent to distinguish between good and bad moves. The current reward function, which also incorporates information about the average tilt angle and, therefore, returns values that are more diverse.

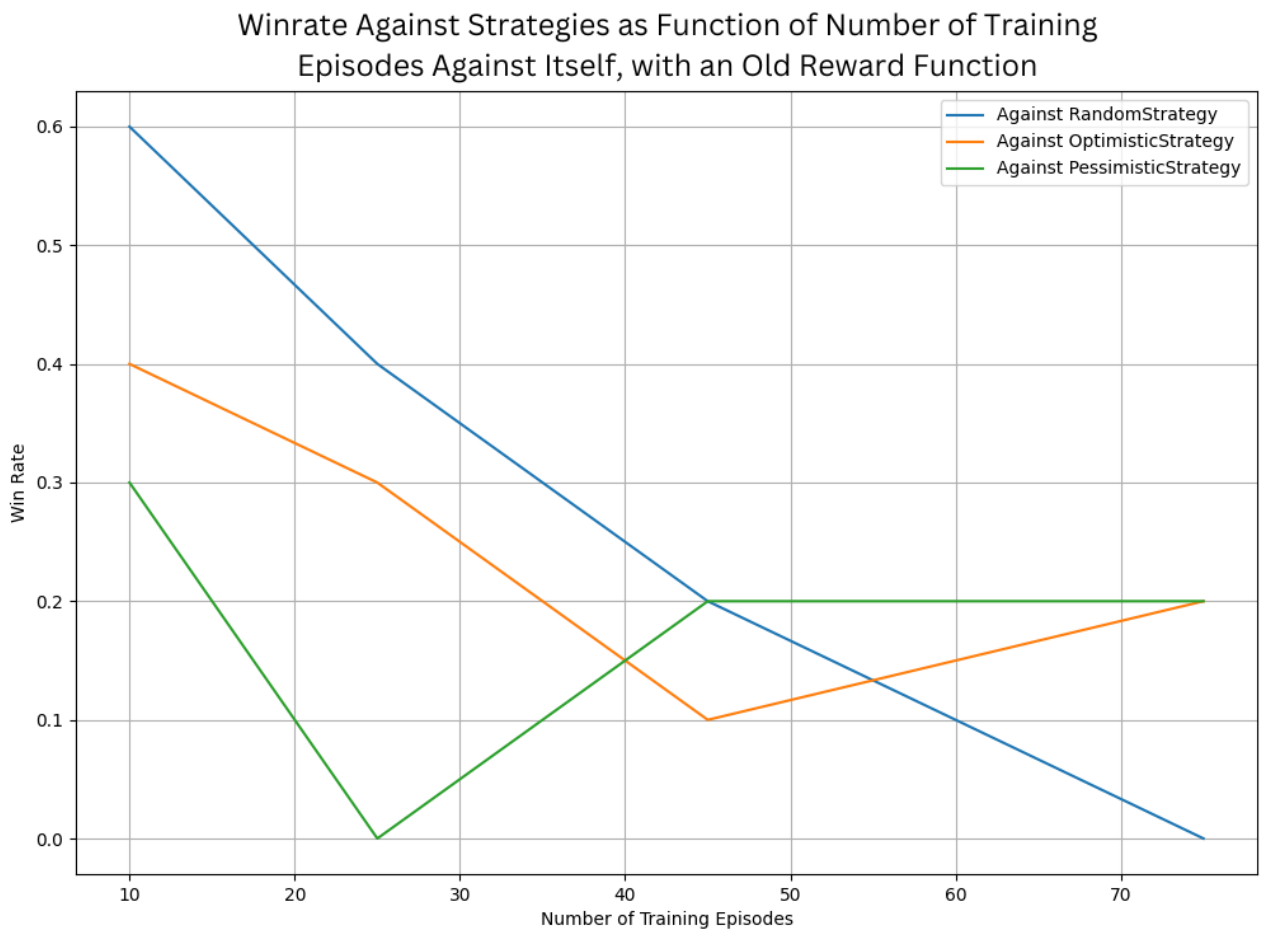

Figure 8: Example how a non-diverse reward function leads to lack of learning

Now we will discuss the performance of the agents under the metric of average moves until collapse. The following plot depicts the average number of moves per game of the Hierarchical DQN agent after playing 20 games against adversaries with baseline strategies as function of number of training episodes against itself from the previous training iteration.

We can see that there is a sudden dip in the average number of moves after 20 training episodes, which is due to the fact that the only best options known to the agent are too risky. This dip corresponds to the low results in the win rate graph after the same number of episodes. We have to note that we created other such plots to ensure the consistency of the results and in some of them instead of a dip there was a big spike. This simply depend on the

random moves the agent explores first: sometimes they are mostly risky, sometimes they are mostly cautious.

Between the episodes 20 and 30 there is a sharp rise in the average number of moves, which is due to rapid balancing out of the riskiness of the moves. After drawing parallels with the win rate graph, we can also see a steep increase in performance there. However, after the spike there is a decline in the number of average moves. If we look again at the win rate graph, we will not see any deep decreases after 30 training episodes. So, likely, this decline means that the agent previously decreased the riskiness of the moves too rapidly, and so the tower fell after a bigger number of moves. But after that the agent gradually started increasing the riskiness of the moves back, which made the games shorter on average.

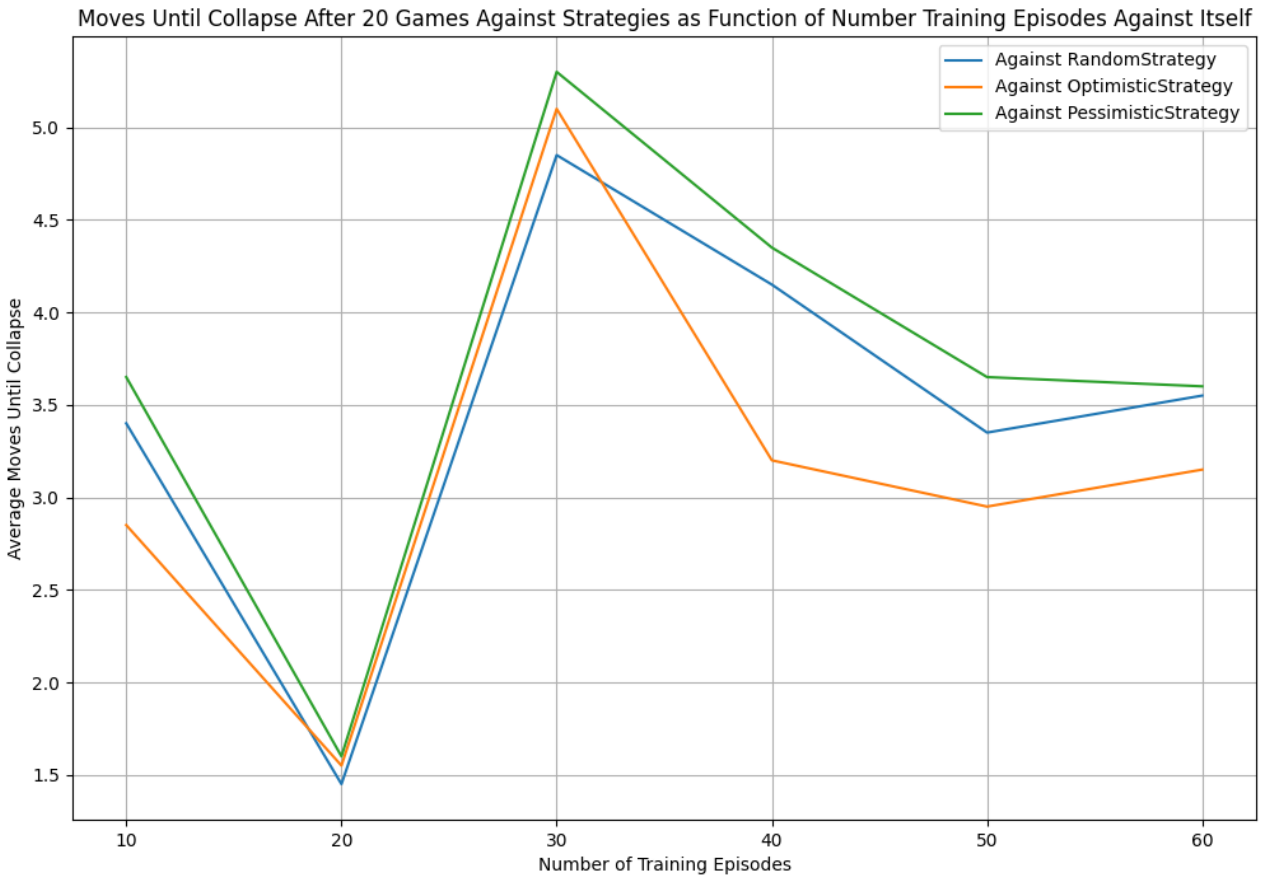

Figure 9: Average number of moves per games after 20 games between Hierarchical DQN and different strategies

Following is the average number of moves per game for the Hierarchical SARSA agent in the same setting as in the previous plot. Every aspect of the plot suggests that SARSA is more conservative than the regular Hierarchical DQN agent. First of all, it starts with a higher average number of moves for all the strategies, which means that it favors safer moves even

in the early stages of learning. Secondly, it adjusts the riskiness of the moves gradually rather than dropping it fast and then slowly bringing it back like the regular DQN does.

While the regular Hierarchical DQN manages to finish a game in approximately 3.5 moves average for all the strategies, the SARSA agent takes more moves for Random and Pessimistic strategies. Although we can speculate that eventually SARSA would adjust the Q-values of the actions and the average game duration will go down (we can see that it was on decline after 60 training episodes), but it can also stay this way due to the cautious nature of SARSA as it manages to survive for a few more moves. In any case, we can notice that the difference between the two agents in terms of average number of moves per game is not this big.

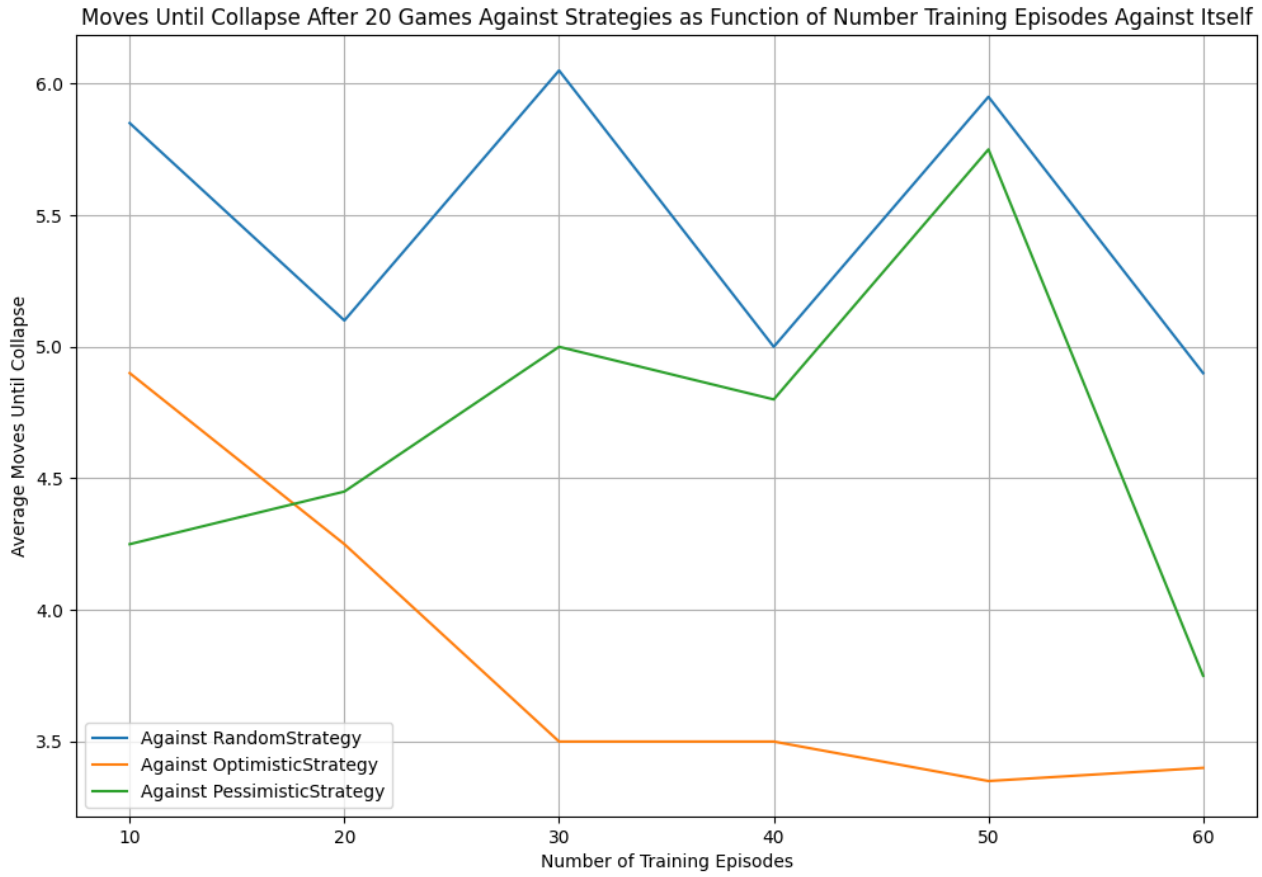

Figure 10: Average number of moves per games after 20 games between Hierarchical SARSA and different strategies

To summarize our findings, the Hierarchical SARSA agent performed best, as it showed the highest win rate at the end of all training episodes, as well as similar to the regular Hierarchical DQN agent average number of moves per game. In general, it is more suited for Jenga than the regular Hierarchical DQN agent because it changes the Q-values at a slower pace, and so it is less affected by the stochastic nature of Jenga, whereas the regular DQN

agent has to update the Q-values drastically and frequently. On the second place there is the regular Hierarchical DQN agent. Finally, on the third is the GSBAS agent that achieved the lowest win rate and is slow, too cautious and unfair for its opponent.

Summary

In this project, the goal was to find a solver for Jenga. Due to the physical nature of the original game, noise is introduced to players' strategies, no matter how good they are. Therefore, we aimed to solve a version of Jenga that is simulated by the Unity physics engine, with the tower shortened for simplicity and without returning the blocks to the top. The objective of the agents is to remove blocks from the tower, so that the tower wouldn't collapse during their turn, but significantly increasing the chances of it falling for the adversary.

To remove a block, agents written in Python send a command to Unity, which executes it and returns feedback on whether the tower collapsed, along with data needed to compute the reward. Agents can choose a block and make an action by specifying the number of level that block is in and the block's color.

A few key assumptions were introduced:

- We simplified the problem by reducing the tower height from 18 to 12 levels and removed the need to return a block on top of the tower. If we define the original problem to be solving a simulation of the Jenga that maximally resembles the real-life game, we expect that the first abstraction would not make the agent perform poorer for the original problem, whereas the second abstraction would because the agent would need to consider how placing a block on top of the tower affects future stability.
- We assumed that the tower was assembled perfectly. Furthermore, we assumed that when removing a block, it doesn't touch any other blocks and just vanishes. We can speculate that both the first abstraction and the second relaxation would make block removal process noisy to the point of not being able to make a successful move for the original problem without some probing mechanism.
- Finally, we assumed blocks in the same level and blocks of the same color can be grouped together. That is, we can combine the level and color of different good moves into an optimal action. This assumption can result in suboptimal moves for the original problem. Actually, this assumption could be relaxed, and we will discuss this later.

The state of the game is represented using two screenshots of adjacent sides of the Jenga tower combined into one image. This image is then converted to greyscale, normalized and

given as input to the agents. While helping with spatial understanding, image usage makes most of the agents highly sensitive to input variations.

We implemented three agents: Greedy Simulation-Based Action Search agent (GSBAS), regular Hierarchical DQN agent and Hierarchical SARSA agent. The SARSA agent performed best out of all agents, making the most cautious moves. We think that the following other algorithms could also work well for this problem: PPO (due to the continuous nature of the action space with different results in different games for the same action) or Evolutionary Strategies (since it is effective against noise).

Initially, we wanted to implement Monte-Carlo Tree Search instead of GSBAS, but we realized that the simulations below one level are costly and unnecessary, and so MCTS evolved into a GSBAS, which is a greedy agent that chooses one action that maximizes the reward function out of ten random actions it simulates in real time. We believe that by running simulations in real time this agent has an unfair competitive advantage.

The other two agents are called “hierarchical” because they utilize two neural networks to choose the level to remove the block from and the block’s color separately. After completing the project, we realized that we could enumerate all the possible actions (maximum 36 at any time) and use only one network instead of two. Then we could also discard the last assumption we made. However, we do think that the hierarchical approach would be considerably faster compared to using one network.

Our solution can be generalized to other problems involving the physical simulation of a tower (e.g., building structures or simulating earthquake impacts). The two hierarchical models could also be applied to problems involving multi-level decision-making.

We hope you enjoyed learning about our project! Thank you for your attention.