Hello, our names are Alon Biner, Mariia Makarenko and Yonatan Vologdin, and today we will present our project for the course Introduction to Artificial Intelligence. In our project we tackle the challenge of solving the game of Jenga using various AI algorithms.

First, we will describe the rules of Jenga for those not familiar with the game. In the original version of the game there are 54 wooden blocks. To start the game, these blocks are stacked in levels of three on top of each other, so that the levels alternate direction. So, in total there are 18 levels in the initial tower.

Show physically

After building the tower, the players make moves in turn, and the person who stacked the tower plays first. Every move consists of taking one block from any level of the tower (except the one below an incomplete top level) and placing it on the topmost level to complete it. Players can touch the tower with only one hand at any time. The game ends if either the tower falls either completely, or if any block falls from the tower, and the loser is the player who ended the game.

Show physically

The Jenga game is challenging for the AI player, because in order to win the player needs to make a move that balances out the risk of the tower falling: in needs to be safe enough for the tower not to fall in this move, but risky enough to increase the chances of the tower falling during the opponent's move.

One of the main challenges for developers of Jenga AI solver is the physical nature of the game. That is, simple inaccuracy in pulling out blocks can make the player lose, independently of how good their strategy actually is. Therefore, we made a few key assumptions and relaxations.

First, we assumed that the blocks are removed perfectly. That is, they just vanish from their places without touching the other blocks. Second, we relaxed the problem to the total of 10 levels and removed adding the blocks back on top of the tower after pulling them out. And there were also other assumptions that we will discuss a bit later.

Using all these assumptions, we created an environment in Unity, so that the Unity physics engine controls the positions of the blocks after each move. The architecture is as follows: our AI agents take information from the python environment object, which in turn sends and receives data to and from Unity via TCP. When the agent wants to make a move, it selects the level and the block at that level, sends a command via the environment object, the environment object sends the command to Unity, and the block is removed.

All the AI agents get rewards for their actions, so that the rewards are computed in the same way for all of them. And they try to make actions that maximize the reward to improve their performance. Since we want our agent to make the life harder for its opponent, we give bigger rewards for actions that we assume decrease the stability of the tower for the opponent but aren't very risky for the player themselves. The lower the level of the block pulled out, the better, and the more the tower was swinging during the move the better, if the tower didn't fall in the process. To calculate the swinging, we compute the maximum average incline of the blocks from the center of mass of the tower after the move. If the tower fell, we give to that action a large penalty.

Show physically

As per the algorithms, we tried three different solutions. The first solution is using the algorithm that we called Greedy Simulation-Based Action Search (GSBAS), which basically tries out different possible actions and chooses the one that gives the better reward. However, we thought that it was unfair for the other player that the greedy player knows in advance during which actions the tower falls and avoids them.

The other two solutions are more fair and they estimate how good the actions are during the training process rather than in the game itself. The first such solution is Hierarchical Deep Q Network. This algorithm uses a neural network to estimate the Q-values of actions. It is hierarchical because we split the choice of the best level to take the block from and the best block in level, assuming that they can be chosen independently from each other. This is done this way to speed up the computation.

The second solution is similar to the first one and is called SARSA (State Action Reward State Action). It updates its Q-values based on the action the agent *actually* takes, according to its current policy. SARSA's update rule considers the action that the agent follows in practice, which includes exploratory actions. This leads to safer exploration. On the other hand, DQN uses a target policy, which is greedy, to update Q-values regardless of the action the agent actually took. It uses the maximum Q-value from the next state (i.e., the best possible action in the next state) for the update, even if that action wasn't the one taken in practice.

We chose to represent the Jenga states for the neural networks as a matrix loaded from the picture that is a two-sided "scan" of the Jenga tower. This is to deal with the fact that there are 2^32 possible states in Jenga if we represent them as binary strings with 0 and 1 for each block, depending on whether they are present or not. And the dimensions of the images are smaller.

Show physically

When we trained our algorithms, we created a menu in Unity that allows the user to choose the algorithms to compete against each other and watch how they perform. It is also possible for the human player to play against one of the algorithms, and two humans to play against one another.

We hope that you will try the game out and tell us what you think. Thank you for your attention!