1 Problem Statement

Our project is to implement an agent that learns to play Tetris in an adversarial environment. The Tetris game was invented by Alexey Pajitnov in 1985. The version we use is a 2-dimensional game that consists of a board with a fixed size and a sequence of blocks of different sizes. Whenever the game provides a new block, the player has a certain amount of time steps to place the block on the board, on top of the blocks that have already been placed before. As soon as the block is placed, a new block is provided, and so on. If the new block creates a line across the width of the board, that line disappears and typically the score goes up. The game ends when the board is filled, which means that no new block can be placed on the board without the height of the highest column being larger than the height of the board.

There are seven types of blocks (also called tetrominoes), that are shown in Figure 1. The distribution of blocks is unknown to the player and can be fixed or it can change according to the configuration of the board. In the latter case, the game is adversarial if the distribution of blocks assigns lower probability to blocks that would get the player a higher score if he or she was to place them in an optimal way. In our setting we considered the possibility that we might face such an adversarial environment.

![Figure 1: The different types of blocks used in the game Tetris](image-url)
1.1 General Setting

As part of the RL-Competition, our agent uses RL-Glue to interact with the environment. The agent’s performance is tested by running a series of experiments. An experiment has 4 parts:

- The environment: represents the domain (Tetris, in our case). Each time the agent takes an action, the environment responds by sending the reward and the observation (new state) to the agent.
- The trainer: controls how a particular experiment is run.
- The agent: this is the only part that we have control over and that we can change.
- RL-Glue: the middleware that facilitates the communication among the different components.

In the observation the agent gets from the environment, there is information about the size of the board, the tetromino that is currently falling and the current configuration of the board. Using that information, the agent can decide what is the best way of placing the tetromino on the board. There are not a lot of assumptions we can make about the way the agent will be tested during the competition, but one thing we can assume is the format of the observation we are getting from the environment. The observation contains the size of the board, the piece that is currently falling and the actual board.

1.2 Challenges

The problem we were facing is a challenging one. We need to learn to act well in an environment that has a large number of states. One of the most popular methods in reinforcement learning is using a value function (the sarsa approach), that assigns to each state a value reflecting the expected reward for that state. Each time the agent acts in the world, it gets a reward and goes to a new state. Based on the reward it is getting in a certain state $s$ by taking a certain action $a$ available in that state, it can update the value of the state to more accurately reflect its expected future reward from that state. So in order for the agent to predict the expected value of each state with a certain accuracy, it needs to visit each state enough times to have the experience it needs in order to be able to make accurate predictions about the expected rewards in each state. With a small number of states, the agent is able to predict the values of the states after a manageable amount of time, which is dependent upon the total number of possible states.

In our case, the number of states is very large. If $w$ is the width of the board and $h$ is the height of the board, each square on the board can be either empty or filled. We get a total of $2^{wh}$ possibilities. An average sized board has a width of 6 and a height of 16, which gives us $2^{6\times16}$ states (almost $10^{10}$). We are expecting the width of the board to be smaller than 20 and the height to
be smaller than 40, but this assumption does not make it tractable to apply the sarsa family of algorithms to Tetris, because the state space is really large for any size board and an algorithm that would attempt to learn the value function for each possible state would take too long to converge.

Another challenge we had to face was the adversarial property of the environment. In order to act optimally we could try to estimate the distribution of the tetrominoes we were getting. If the distribution were constant, we might have tried to estimate it by counting the tetrominoes we were getting and then trying to fit a distribution to our samples. We might also have tried to place the blocks in such a way to break multiple lines at a time, since we knew that there could be a possibility that breaking multiple lines at a time might get the player a better reward than breaking the same number of lines in different time steps. The instructions we got stated the following: "the distribution of piece types is not uniform. For example, in some games, there may be an abundance of "Z" pieces, in others, there will be a shortage. But you may expect that the probability of getting one will be lower when you really need one...". So another challenge was to implement the agent in such a way that it would perform well in such an environment. We explain how we dealt with this problem in Section 2.3.

A third challenge was the large branching factor created by the lookahead technique we have used. We are addressing that issue in 2.4. In some versions of the game, the player knows what is the next tetromino it is going to get, and not just the current one. In our case, only the current tetromino is known so the tree produced by a lookahead technique has a high branching factor, making it intractable to look more than 3 steps ahead, even when pruning a large percentage of the nodes in the tree (since some choices of actions are obviously bad).

2 Our Approach

2.1 Function Approximation. Features Used.

To deal with the large state space, we have used a classical approach in reinforcement learning: function approximation. When using function approximation, we need to choose a set of features that characterize the state in general. When we are given a certain state, we can figure out what are all the different features for that state. The value of a state is a weighted sum of the values of the different features for that state. This way, instead of dealing with a large number of states, we only need to deal with a small number of features and their weights.

When using function approximation, the challenge is to narrow down the number of features used so that learning would be tractable (the fewer features we use, the faster the learning). On the other hand, we need to choose enough features so that states that are different get different values. Otherwise, we might not be able to differentiate between states that might give us very different performances. For our agent, we have used the features suggested by Paul
Dellacherie, cited by [2].

The features we used were the following:

- **Number of holes**: The total number of empty spaces on the board that have at least one occupied space above them
- **Number of lines broken**: How many lines were broken on the previous step
- **Well count**: This feature is illustrated in the left part of figure 2. The penalty for a well grows quadratically to the size of the well
- **Number of row transitions**: For each row, this feature records the number of times there is a transition between an empty cell and an occupied cell as you travel from left to right (see 2)
- **Number of column transitions**: This is the same as row transitions, only on the vertical dimension.
- **Individual column heights**: For each column, what is the height of the stack of occupied cells on the board
- **Height differences between adjacent columns**: For each pair of adjacent columns, record the difference in height between them.

The total number of features used is \(4 + 2 \times \text{boardwidth}\).

Figure 2: The ‘well count’ feature (left) and the ‘number of row transitions’ feature (right)

To learn the weights for these features, we have used the cross entropy method. This method was successfully applied to Tetris by [1]. We present this model in the next section.

2.2 The Cross Entropy Method

It has been shown in previous research and competitions on the Tetris problem that an evaluation function based on weighted features outperforms other methods directly derived from reinforcement learning, especially when the weights are previously learned beforehand [1]. This method is typically used when it is both desirable and plausible to estimate the value of a state that can be
compactly expressed with a number of numeric features. The learning problem consists of learning the weights that best approximate the true scores for the state of an environment.

Cross-entropy works as follows. Before starting to learn anything, identify good basis functions that capture relevant features about the environment in question. First the master weights are initialized to some values so to ensure good performance in the long run. Zeros are a good start because any weighting can be reached from that point. Next draw a number of sets of weights from a distribution that is based on the master weights. According to [1], a Gaussian distribution works for many domains. With each set of weights, play a single game and record the score. When completed, use the mean of the weights that scored the highest as the new master weights. Repeat this process.

However, a game like Tetris, which may give very different scores depending on the sequence of blocks that arise, may have problems with converging too quickly to a non-optimal setting. To counter this possibility, it is useful to add some noise to the variance of the Gaussian used to draw the weights. Like a simulated annealing algorithm, this noise can decrease over time until the weights have converged to a more optimal setting.

2.3 Probability Estimation

The prerequisite of looking ahead is that we need to know the type of blocks we will get from the environment. This information is not available to the agent, but can be estimated. As discussed in the previous sections, we are expecting that the distribution of tetrominioes not be fixed, but will depend on state of the game. The specific dependence can be estimated while playing and keeping track of the frequencies of blocks under specific configurations of the board. In this section, we discuss the way we are estimating the probabilities of the tetrominoes that we are getting from the environment.

Let \( P(b_i|S) \) denote the probability of getting a block of type \( i (i = 1 \text{ to } 7) \), the current state being \( S \). For this specific state, we can keep track of the frequencies of each block type. We have that \( P(b_i|S) = \frac{\text{count}(b_i|S)}{\sum_{j=1}^{7} \text{count}(b_j|S)} \). An appropriate choice of the state representation is crucial for estimating meaningful probabilities. As we have pointed out before, it is intractable to have the current state be the entire board configuration. Considering the adversary Tetris scenario, the coming block is unlikely to be what the agent needs so that it can eliminate lines for higher reward. So it would make more sense to model the state in a way that would capture how much we would need each type of block in the current configuration of the board. One way to represent the state \( S \) would be a 7-element tuple: \( S = < L_1, L_2, L_3, L_4, L_5, L_6, L_7 > \), where \( L_i \) is the maximum number of lines that can be broken on the current board by block type \( i \). The values of \( L \) are \(< 4, 2, 2, 2, 2, 3, 3 > \) (corresponding to each block type).

Once we have the block probabilities, the expected value of given board configuration in k-th step \( V_k = \sum_{i=1}^{7} P(b_i|S)V_{ki} + \text{Reward} \). \( V_{ki} \) is the value of
state after dropping block of type $i$.

### 2.4 Look Ahead

We have taken the approach of searching ahead a small number of steps to get a better picture of what placing a piece in a given position several turns from the current time step. We then take the expected value of those next states given our current estimate of the transition probabilities. In addition, to save time we opted to prune away some of the branches that have values that are too low to consider for the next action.

In many ways, the lookahead is at the heart of our approach to the altered version of Tetris. Because the next block probabilities depend on the state of the board, the value of a board cannot be approximated by basis functions alone. For example, perhaps by cross-entropy methods the features of a certain board have low values, but the transition is such that the next piece is likely to be very harmful. Traditional approaches do not explicitly account for this possibility, but it is the problem we face in the adversarial Tetris domain.

Consider the pruning ratio to be $p$. The specific method is to check the value of the board $b_i$ produced by each of our $m$ possible moves given the current tetromino, by the evaluation function consisting of our weighted basis functions. For any block, $m = (\text{width})N_{\text{rotations}}(\text{block})$. These moves are then ranked, and the bottom $(1 - p)m$ moves are pruned away. Then, the boards $b_i \in [1...m]$ produced by those moves are assigned to a probability distribution based on previous experience corresponding to pieces. This process is then repeated for each $b_i$, and for each piece according to the assigned distribution. The new value assigned to the original move is the expected value of these next round evaluations. Although searching ahead many steps would lead to the best possible estimates, we opted to stop after 2 or 3 steps, depending on the board size. As implemented, the method amounted to a blend between depth first search and breadth first search.

There is a trade-off to consider when answering the question about how many actions to prune. Searching fewer actions speeds up the computation, but may obscure some good actions. There is also a different number of actions for every piece. Consider that the pruning method is set to look ahead $d$ steps and searches the top $p$ percent of branches. In general the increase in computation of looking ahead one more step while searching half the number of branches is found to be $c = \frac{7m}{2w}$, where $m$ is the average number of moves per block. We settled on $p = \left\lceil \frac{12}{w} \right\rceil$, so that $p \cdot m = N_{\text{rotations}} + 1$, where $w$ is the width of the board.

Another more sophisticated technique involves greedily searching the best moves up to a predefined depth, without looking down less good moves in parallel. In other words, update the value of the currently estimated best action in a way similar to the pruned lookahead. If that action remains the best value after this update, search it again, this time looking at the best actions for each next piece. If at any step the expected value of the leaves becomes less than the next best action, then search that next action instead. In this way the algorithm is
always getting more accurate estimates for its best actions, and therefore will be better able to distinguish between those actions. The algorithm can search ahead an expanded number of steps, because this method reduces the base of the exponent to the number of pieces, which is seven. After some preset number of board evaluations, the searching can stop. We had little success with this method as it was difficult to ensure that the values were not always worse after adding a new piece. We show here that a well-designed lookahead process can boost performance. Future advances for this problem will be inspired by planning methods used in RL, such as knowing which states and actions to explore and which have been explored enough.

2.5 Crazy Tetris

For the competition, the agent must take 5 million time steps on each MDP. On boards with large width, starting with some simple weights and training until good weights are found can take much longer than the allotted amount of time steps. Because of this, we’ve made the agent store the resulting weights at the end of every MDP run and have trained it on all the learning MDPs.

The algorithm has two parts: an part where it learns weights by playing 5 training cycles on each of the training MDPs (this is the offline learning part: once the competition starts, the agent has already learned and stored these weights and is able to choose the weights that best fit the configuration it’s facing) and a part where it’s learning as it interacts with the new environment (online learning).

**Offline learning:** given the 20 different training MDPs we had access to in the training phase of the competition, we’ve applied the cross-entropy method to learn the weights for each feature for each of the MDPs environment. The 20 environments are grouped according to the width of the board.

**Online learning:** When the agent starts on a new MDP, it loads all of the stored weight sets with the right amount of weights (recall that the number of weights is dependent on the board width). There are three possible cases here.

- **Case 1:** No stored weight sets with the right amount of weights are found. In this case the starting weights are set to those from the Dellacherie algorithm and the weights are all given a high initial variance.

- **Case 2:** One stored weight set with the right amount of weights is found. In this case the starting weights are set to those from the stored weight set and the weights are given no initial variance.

- **Case 3:** Multiple weight sets with the right amount of weights are found. In this case 30 games are played using each stored weight set and the starting weights are set to those from the stored weight set that performed the best. The weights are given no initial variance.

After the starting weights have been set, the cross entropy method continues as normal. For the competition MDPs, the noise decreases linearly and hits 0 at 2.5 million time steps.
3 Results

The chart below (see Figure 3) shows the performance of several methods on the first training MDP. The Depth 0, Depth 1, and Depth 2 data sets were from the first phase of the project, and the new Depth 3 With Pruning And Loaded Weights dataset was from the second phase.

![Performance chart](image)

**Figure 3:** Performance of the agent using different parameters.

One thing to notice is that agent that uses the new dataset performs the best out of all the agents we have tried. This agent starts off higher because of the loaded weight set. These weights were not from a training session on this MDP, but the Depth 2 dataset catches up over time. The new dataset does improve performance beyond that of the loaded weights. This suggests that evaluating all the stored weight sets and playing the whole 5 million time steps with the stored weight set that had the highest performance may be a valid strategy for a competition MDP.

Another thing we have noticed was that the agent seems not to evolve too much and do a lot of learning. In reality, its average reward does go slightly up, which is hard to see because of the large variability in its performance.

This result is significant, because the MDP we tested it on (also called Parameter 0) was adversarial. This fact was confirmed by the organizers of the competition and also by the following observation: we ran the Dellacherie algorithm for about 30 games on the standard (equal piece probability) Tetris environment of the same 6x16 board size and it got about 155 lines per game. On parameter 0 it gets about 30. This large difference is due only to the difference in the pieces that drop. Therefore Parameter 0 is adversarial.
Below (see Figure 4) we have another graph that illustrates the performance of our agent on the proving MDPs, compared to the Test Agent that the organizers provided (which in this case is the standard agent that doesn’t do any learning) and also compared to the upper bound on the reward that could have been gotten on the proving MDPs (the upper bound was provided by the organizers of the competition).

![Total Reward Graph](image)

Figure 4: Performance of our agent compared to the standard agent (‘Carlos’ Test Team) and the upper bound on the total reward

4 Conclusions and Future Work

The project we are participating in is a challenging one. Coming from a Reinforcement Learning perspective, we are facing a problem that is difficult to solve (shown to be NP-complete), with a large state space and an environment that, even though markovian, changes the distribution probabilities of the tetrominoes it is providing in a way that makes it harder to both predict the distribution of pieces and try to get as high a reward as possible.

To face these challenges, we have combined a few methods that have successfully been used before in Tetris or in Reinforcement Learning problems that were of a similar nature as Tetris. We adopted the cross-entropy method to estimate the weights for the features we have used in our approximation of the value function. The cross-entropy method has allowed us to learn the weights in a reasonable amount of training cycles. We were also able to store these weights and use them later, when faced with a new MDP. In order to predict the blocks, we have used a probability estimation that is dependent on the configuration of
the board, taking into account that the environment might be adversarial. This probability estimation also helps us evaluate the next states in our lookahead technique. The lookahead is probably the component that boosts the performance the most, even though the loaded weights of the features play a very important role in that too.

Our future work includes looking into how we can improve the performance of our agent even more. One area where that might happen could be in increasing the steps of lookahead. With the parameters we have been using, three steps has shown to be a practical limit to the number of steps we look ahead. We are interested in trying to find a more efficient way of pruning the tree so that we would be able to look ahead more. We could also try to run our agent on a more powerful machine where it wouldn’t take the agent too much to come up with its action on each step (right now, with more than 3 steps of lookahead, we are estimating that we would be going over the recommended time for running the agent in the competition). We think that the one promising area of improvement would be to increase the amount of steps the agent is looking ahead. Another major challenge is to better learn the probability model that the MDP uses to emit blocks.

References
