Pentomino Tetris AI: A Reinforcement Learning Approach

Simon Ingemann Axelsen

[1] Aalborg University Copenhagen, A. C. Meyers Vænge 15, 2450 København SV, Denmark
aau@aau.dk
[2] Springer-Verlag GmbH, Tiergartenstraße 17, 69121 Heidelberg, Germany
lncs@springer.com
http://www.springer.com/gp/computer-science/lncs
[3] Ruprecht-Karls-Universität Heidelberg, Grabengasse 1, 69117 Heidelberg, Germany
saxels22@student.aau.dk

Abstract. An implementation of a reinforcement learning (RL) agent for playing Pentomino Tetris, a variant of Tetris using pentominoes (shapes made of 5 connected squares) instead of the standard tetriminoes.

Keywords: Pentonino Tetris · Reinforcement Learning Approach · AI.

1 Introduction

Tetris is one of the most recognizable puzzle video games made, known for its simple but engaging game-play. This mini-project focuses on a variant called Pentomino Tetris, which increases the complexity by using 5-square pieces (pentominoes) instead of the traditional 4-square pieces. The increased complexity of the piece shapes will create a more challenging environment.

The goal of this mini-project is to develop an AI agent capable of playing Pentomino Tetris using reinforcement learning techniques.

2 Theoretical Foundation

Reinforcement learning (RL) represents a field in machine learning where an agent learns to make decisions by interacting with the environment. Unlike supervised learning, which relies on labeled data, or unsupervised learning, which finds patterns in unlabeled data, reinforcement learning focuses on learning best behaviors through trial and error. The agent receives feedback in the form of rewards or penalties based on its actions over time, thereby training its decision-making strategy to maximize the cumulative reward. The components of a reinforcement learning system include an agent, an environment, states, actions, and rewards. The agent observes the current state of the environment, selects an action, and receives a reward signal and then transitions to a new state. This

creates a feedback loop that makes the learning process, where the agent's goal is to create a mapping from states to actions that maximize the cumulative reward.

For the Pentomino Tetris, the agent controls the movement and placement of the Pentomino pieces, the environment is the game grid, the states represent the configuration of the blocks on the grid, and actions include movement like left, right, rotate, and soft or hard drop. The rewards are determined mostly by successful line clears, but also by survival time.

2.1 Q-Learning

Q-Learning forms the foundation of this approach to try to solve Pentomino Tetris. Q-learning is about learning a function that estimates the value of taking a particular action in a given state. The "Q-function" helps the agent decide which action is most beneficial in the situations that it encounters.

2.2 Linear Function Approximation

In reinforcement learning problems, such as in games like Tetris, the state space is exceptionally large. The traditional tabular approaches that store values for each state-action pair in a table quickly become impractical due to memory constraints and the limitation of visiting every possible state during training. Linear function approximation addresses this challenge by representing the value function as a linear combination of state features, and estimate values without needing to remember every possible state. Instead of tracking each situation individually, it look at few important elements about the stat called "features".

2.3 Epsilon-Greedy Exploration

A challenge that is vital in reinforcement learning is balancing exploration (trying new actions to discover potentially better strategies) with exploitation (using known good strategies). The epsilon-greedy strategy provides a solution to this exploration-exploitation dilemma. In epsilon-greedy policy, the agent selects the action with the highest estimated value with probability $1 - \varepsilon$, and chooses a random action with probability ε. As the model is training, the ε is gradually decreasing and shifting the agent's behavior from mostly exploratory to more and more exploitative.

In the case of this Pentomino Tetris, the standard epsilon-greedy approach has been modified slightly with the goal of trying to make the agent learn more effectively in the state space. Rather than using a uniform random exploration, a probability-weighted exploration strategy based on historical performance has been developed. Each action receives a probability proportional ($\propto$) to its historical success in clearing lines.

$$P(a) \propto 0.1 + 0.9 \times \frac{\text{number of times action } a \text{ led to line clears}}{\text{number of times action } a \text{ was taken}}$$

It is then normalized to ensure all the probabilities sum to 1. This approach ensures that even during exploration, the agent favors actions that have previously led to line clears, aiming to make the learning process faster.

Additionally, an exponential decay for ε, starting at 1.0 (pure exploration) and gradually decreasing to 0.05 (95% exploitation, 5% exploration) over the course of the training.

3 Implementation

3.1 Game Environment

The game environment for Pentomino Tetris consists of a $10x20$ grid playing field and five distinct pentomino shapes (I, L, P, T, and V). The gameplay mechanics follow traditional Tetris rules: pieces fall from the top of the grid, can be moved horizontally, rotated, and dropped. When a row is completely filled with blocks, it is cleared, and all the blocks above shift downward. For the reinforcement learning, a `GameWrapper()` class has been made to mediate the interaction between the agent and the game environment. It provides

- **State representation**: Converting the game state to capture strategic elements.
- **Action space**: Defines the five possible actions (move right/left, rotate, soft/hard drop)
- **Reward structure**: Calculate rewards based on game events and board configurations.
- **Episode management**: Handles game resets and tracks statistics.

3.2 Q-Learning implementation

In the implementation of Q-learning in Pentomino Tetris, the agent learns the value of different moves (left, right, rotate, soft drop, hard drop) for various game states. This learning process happens iteratively through its interaction with the environment, and after each move, the agent updates its understanding based on the reward received.

```
q_current = np.dot(self.weights[action], state[0])
q_next = 0 if done else np.max(self.predict(next_state))
target = reward + self.gamma * q_next
error = target - q_current
```

This snippet from the `learn()` method shows how the Q-learning has been implemented: the agent compares what it expected to happen `(q_current)` with what actually happened `(reward + future value)` and adjusts its knowledge accordingly.

3.3 Linear Agent Implementation

Because the number of possible board configurations is too large for it to store and learn values for each individual state, the linear function approximation has been implemented. It offers the solution of representing the value of a state-action pair as a weighted sum of important features. In the approach used in this implementation, key attributes of the game state that influence the decision quality have been identified and categorized; these features that capture strategic elements of the game have been extracted, and the agent learns how much weight to give each feature when making decisions. Each feature represents a strategic element of Tetris gameplay:

```
features = np.array([
    normalized_holes * 2.0,
    normalized_bumpiness * 1.5,
    normalized_height,
    complete_lines * 3.0,
    near_complete_lines * 2.0,
    wells * 0.5,
    avg_height / GRID_HEIGHT,
    center_distance,
    piece_value,
    nearly_nearly_complete_lines
])
```

When initializing the weights of these elements, the agent is given a sort of head start by having human knowledge and expertise weighing the different elements. A snippet of some of the weights that are given can be seen here.

```
#Holes:
self.weights[:, 0] = -1.0
#Bumpiness:
self.weights[:, 1] = -0.5
#Complete lines:
self.weights[:, 3] = 3.0
```

A strong negative weight has been given to holes and a strong positive weight to complete lines (the primary objective). And as the agent is playing, it refines these weights based on its experiences, as seen here.

```
#Update weights
self.weights[action] += learning_rate * error * state[0]
```

The goal of this learning process is to shift the weights towards the values that accurately predict which actions lead to higher rewards, and over time the agent should learn to recognize beneficial board positions and avoid damaging ones.

3.4 Training Process

The training process incorporates tuned learning parameters that impact the agent's performance. The implementation uses four key parameters:

```
self.alpha = 0.005
self.gamma = 0.95
self.epsilon = 1.0
self.epsilon_min = 0.05
```

The learning rate (alpha = 0.005) controls how quickly the agent incorporates new information into its policy. The discount factor (gamma = 0.95) determines how much the agent values future rewards compared to immediate ones, so a higher value makes the agent consider long-term consequences of actions rather than immediate rewards. In Tetris, where moves that create immediate rewards might lead to problematic board configurations later.

Reward structure The reward structure is designed to guide the agent towards playing the game using effective strategies, with the primary reward elements for the agent being line-clearing. Line-clearing receives exponentially increasing rewards (2000 for one line, 4000 for two lines, 8000 for three lines, and 16000 for four lines). This steep reward gradient is in place to encourage the agent to pursue multiple simultaneous line clears rather than single-line clearances. To try to minimize problematic board configurations, several penalties have been incorporated. Each hole (empty space with blocks above it) gives a penalty of -2. Stack height beyond the board's midpoint gives progressive penalties that increase with height. Additionally, the agent receives a small survival bonus (+0.15) for successfully placing pieces without triggering game over.

4 Results

The training process for the Pentomino Tetris agent ran for 1000 episodes, generating multiple performance metrics that provided insight into the agent's learning.

The "Score per Episode" plot demonstrates the agent's game performance over time. Notably, the agent struggled to achieve significant line clears, as evidenced by the plot, which shows predominantly zero values with only occasional single-line clears. This difficulty in clearing lines persisted throughout training, indicating that the agent primarily learned survival strategies rather than mastering the more complex task of line completion.

5 Discussion

The linear function approximation approach that is implemented for this model represents a balance between computational efficiency and learning capability.

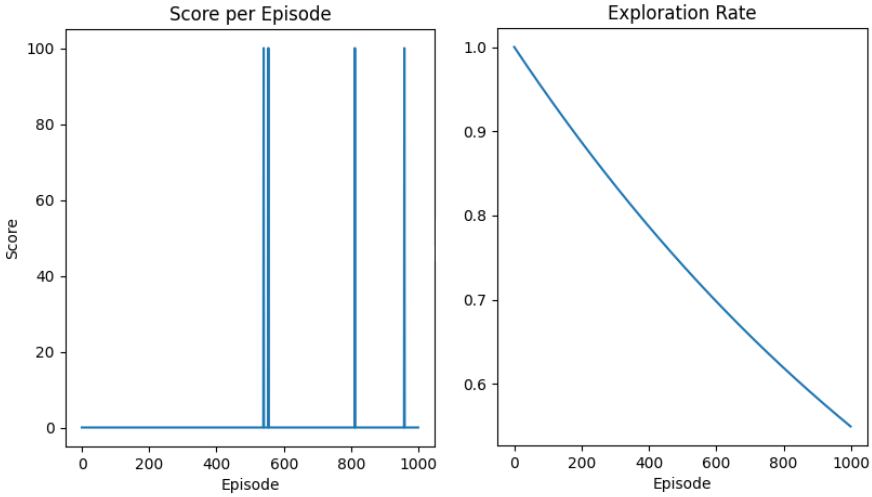

Fig. 1. Figure showing two plots, from left to right: Score per Episode and Exploration Rate

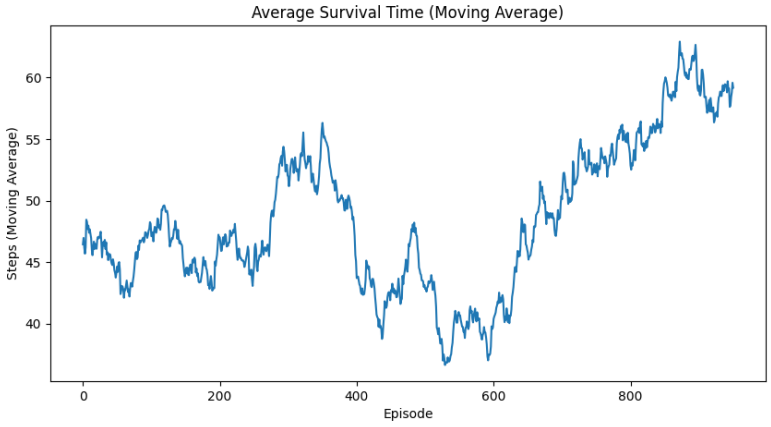

Fig. 2. Figure showing Average Survival Time

The choice of a linear model, instead of more complex deep learning architectures, significantly reduces computational requirements and training time. This efficiency came with tradeoffs, however, most importantly, it likely contributed to the agent's limited success in line clearing.

One fundamental challenge lies in the representation capacity of the linear model. Since Tetris in its nature features complex, non-linear relationships between board configurations and optimal actions, that linear approximation can not fully capture them. The linear combination of the features that was attempted to be extracted, which had the most relevant game characteristics, may not have sufficiently represented the nuanced decisions required for more expert gameplay.

The state space of Pentomino Tetris presents another challenge. With five-block pieces rather than the traditional four-block pieces, the game complexity increases, and likely in the training, only a small part of the possible state space was explored. This limitation, combined with the fact that the reward signal for line clears being relatively rare events, created difficult learning conditions. The agent appears to have learning basic survival strategies but struggled to consistently create and exploit line-clearing.

5.1 Alternative Approaches

An alternative approach to maybe get more consistent results and a better-trained model could have been to train the agent on a traditional 4-node Tetris instead of the 5-node one that it is trained on, therefore having less complexity in its training, leading it to explore more of the space and not get stuck on initial line clearance. The space would then be upgraded to the Pentomino Tetris variant after training, and then see how well it can adapt to the new complexity. Another alternative approach could have been to let a human input manually play some of the first training rounds and give the agent a better baseline for correct actions.

6 Conclusion

Despite the limitations, the mini-project demonstrates the feasibility of applying reinforcement learning to a complex game, Pentomino Tetris. The agent did learn basic survival strategies and showed signs of understanding important game concepts like hole avoidance and height management. With further refinements to the function approximation approach, reward structure, and exploration strategy, or with a transition to more powerful non-linear models, significant improvements in performance could be achievable.

References