A Genetic Approach to the Formulation of Tetris Engine

Jiachen Zhang, Miaoqi Zhang, Shuheng Cao
{j756zhan, m337zhan, s53cao}@uwaterloo.ca
University of Waterloo
Waterloo, ON, Canada

Introduction

- **Motivations**

  With the great triumph of AlphaGo in 2016, more and more researchers are interested in using machine learning techniques to solve more advanced and complicated games such as StarCraft II. However, in the authors' opinion, it also provides us with an invaluable opportunity to revisit some of the traditional video games, where we aim for more reliable and satisfying results. Consequently, **this project's primary motivation is to introduce, implement, and compare several different approaches to tackle the Tetris**, a tile-matching video game with enduring appeal and popularity. In brief, Tetris is a tile-matching video game where the player will rotate and place seven different kinds of Tetrominos on top of each other. Horizontal lines will be cleaned up once they are complete, and a score will be awarded for that. This game's ultimate goal is to achieve as many marks as possible before the pieces reach the top of the game board, and therefore, there is no victory in this game.

  Solving Tetris is a crucial and intriguing topic due to the two reasons below. On the one hand, Tetris is essentially an extraordinary optimization problem because each game will end no matter how well the Tetrominos are placed (Burgiel 1997). As a result, there is no such thing as perfect solutions to Tetris and there is always room to improve. On the other hand, the analyses and comparisons mentioned in this report are not limited to Tetris only, where we could broaden them to real-life problems, such as self-driving cars and robotics, with appropriate modifications on the architectures introduced in this article.

  Finally, solving a relatively simple game like Tetris will help us better understand the related Reinforcement Learning models. One of the main problems the machine learning community faces is the lack of explainability and interpretability for most of the models. The direct analyses on most of the recent models, such as AlphaGo and AlphaZero, are notoriously complicated and challenging, but with a more straightforward and simplified setup like Tetris, it gives us more opportunities to have a more in-depth insight into what is happening under the hood. Consequently, the results from simple setups will contribute to a better and deeper understanding of the more complex models.

- **Methodologies**

  The problem we are trying to solve is to design an agent that could achieve as many scores as possible in the Tetris game. We will implement three algorithms, including the baseline method, as shown below.

  – First and foremost, we built a Tetris interface for both visualization and training purposes. For the sake of communication with our models, the interface is written in Python and it will support two basic functionalities:
    * Generating all the successors for current state.
    * Calculating score or reward for each state.
  – Next, we will solve the problem using a hand-crafted agent. Inspired by Bertsekas and Tsitsiklis's paper (Bertsekas and Tsitsiklis 1996), we will manually chose the weights for 4 most representative state features: the number of holes, the height of the highest column, the height of each column, and the difference in height between consecutive columns. This approach is based mainly on the heuristic search algorithm and involves a lot of trails-and-errors. The result serves as the benchmark for the project using the evaluation metrics mentioned below.
  – On top of that, we will also tackle the problem using local search algorithm. The main idea is to use a genetic algorithm to automatically find an optimal weight combination for 9 state features, where the detailed description for the features could be found in the Methodology section.
  – Ultimately, we will solve the problem with reinforcement learning. The main idea is that we will build and train a deep Q-network (DQN) to evaluate all the successor states of the current state. More precisely, the DQN will receive a bitmap representation of successor and output a non-negative number represents the score for it. After that, the agent will choose the successor with the highest score with probability ϵ and otherwise, it will select a random successor.

  As for the evaluation metric, we will use the number of

lines cleaned up before game over as our primary metric so that we could compare the three methods with each other as well as implementations from other papers. One thing worth mention is that in the DQN training, we won't directly use the evaluation metric mentioned above as the reward because it is too sparse. Instead, we will design a reward that is positively correlated with the evaluation metrics. More concretely, we will use the following rewarding scheme:

- Game over gives -100.
- Cleaning up k lines gives $10k^2$.
- Safely landing a piece gives 1.

Related Work

There is a number of algorithms about Tetris so far. Most algorithms for Tetris use features and a linear evaluation function (Algorta and Simsek 2019). The algorithms define multiple features and assign a weight value to each of the features. For a specific state with an existing Tetromino, it will use the evaluation function to calculate the evaluation value according to the weight of the features for every possible states. And then a "best" placement of the Tetromino will be picked according to the evaluation value.

Tracing back to 1996, J.N. Tsitsiklis and B. Van Roy formulated Tetris as a Markov Decision problem. They introduced Feature-Based Method for Large Scale Dynamic Programming. The algorithm introduced two features, which were the number of "holes" and the height of the tallest column. Each state will be represented as a two-hundred-dimensional binary vector since the Tetris board is form by 10×20 grids. And a seven-dimentional bianry vector will represent the current falling Tetromino since there are seven type of Tetrominos in total. The algorithm can eliminate 31 rows on average of a hundred games (Tsitsiklis and Roy 1996).

Later on, more features have been taken into consideration. For example, peak height, landing height, number of holes, row transition, column transition and eroded piece cells (Wei-Tze Tsai and Yu 2013).

These features were identified by the best artificial Tetris player until 2008 and introduced evalutation function: $-4 \times holes - cumulative\ cells - row\ transitions - column\ transitions - landing\ height + eroeded\ cells$ (Algorta and Simsek 2019). However, most algorithm will perform a row elimination whenever it is possible. This is not optimal in the long term because there could be some state that multiple rows can be eliminated at once (Wei-Tze Tsai and Yu 2013).

Tsai, Yen, Ma and Yu implemented "Tetris A.I.". Scores will be rewarded if a I-Tetromino is dropped and 4 rows are eliminated. The main idea is to make this kind of move as many as possible. The solution for this is to stack on the 9 columns and remain the left column empty. As long as an I-Tetromino apprears, it will be drop at that seperate column to eliminate multiple rows (Wei-Tze Tsai and Yu 2013).

Moreover, they also implemented another model called "Two Wide Combo A.I.", which is a little bit more complex than the previous algorithms. It breaks the process into two parts. One is to "Stack" on the left eight columns by using BFS, and the other part is "Combo", which is to eliminate rows consecutively to earn Combo Bonus by dropping Tetrominos into the rest two columns (Wei-Tze Tsai and Yu 2013).

The use of generic algorithm provides a new way which is worth taking into consideration. In 2006, Szita and Lorincz implemented cross-entropy algorithm. New features were introduced and for each feature, multiple games were played. It took the mean and standard deviation of the best weight of the linear policy that maximize the score and generated a new generation of policies (Algorta and Simsek 2019).

Back in 2003, Thomas Grtner, Kurt Driessens and Jan Ramon introduced a new approach to Tetris optimization(Gärtner, Driessens, and Ramon 2003). What's innovative in this paper is that researchers primarily used Relational Reinforcement Learning (RRL), training the network with Gaussian processes and graph kernels. Relational Reinforcement Learning advances traditional Reinforcement Learning by integrating with a relational learning programming which is also known as inductive logic programming. The use of Gaussian processes is to make probabilistic predictions and allow the agent to incrementally perform learning tasks. On the other hand, graph kernels help represent various states and actions during training process in a more structural way, which greatly facilitates learning. Compared to some previous RRL models which train data based on techniques such as decision trees, this approach has proven to be more effective.

In 2016, researcher at Stanford University continued to optimize Tetris engine performance using Reinforcement Learning (Matt Stevens 2016). They used a notation called Q function. This function evaluate actions taken by the AI at various stages/states during the game; the value of this Q function tells us what the best next action is. Researchers attempted to improve convergence of Q function with the aid of heuristic functions. This method yields the best performance when combined with grouping actions. Basically, grouping actions means the entire sequence of actions a single piece take as it drops to the bottom whereas a single action is just one movement of the piece such as one space to the left. Although grouped actions increase game length compared to single actions, they achieved a much high game score. Another technique that significantly boosts the performance is transfer learning. Transfer learning effectively scores an average of two lines per game, compared to no lines for learning from scratch. The final technique used is called prioritized sweeping. Prioritized sweeping involves calculating a priority value. Based on that value, the researchers sampled actions based on probabilities proportionally to their priorities. This technique solves several problems, for example, the issue that certain actions gotten drastically under-represented in the experience dataset.

Methodology

Handcrafted Agent

Our first approach is a manually trial-and-error approach, or more precisely, we will assign a weight manually for each feature below according to our intuition and experiment results. Then, for each generated successor states, we will always choose the state with the lowest heuristic value, which is calculated based on a summation of the four weighted features. Here are the four features we are going to use and a rationale for choosing them:

1. **number of lines cleared**[1]

   This feature faithfully reflects our evaluation metric since we will achieve scores only by cleaning lines.

2. **number of holes**

   The holes in the Tetris game are particularly harmful because, in order to clean the line containing a hole, we need to clean up all the lines above that hole.

3. **sum of difference of height between adjacent columns**

   The "gaps" in the board make it harder to clear rows simply because the Tetrimonos are generated randomly, and those "gaps" are not likely to be filled perfectly by the next Tetrimono.

4. **maximum height**

   The higher the board is, the closer it is to the "ceiling" (vertical limit of the board), which means there may be fewer steps to play before the game terminates.

We would run simulations consecutively and record the performance associated with the weight combination. Based on the results, we would adjust one or more weight values and use them in the next simulation until the game behaves rationally in human criterion.

An advantage of this handcrafted approach is that we are somewhat informed of what features should be rewarded and penalized. Compared to adjust the weight using algorithms, manually adjustment will give us a better understanding and intuition about the game during the weight adjustment process.

Local Search Agent

Essentially, in a handcrafted agent approach, human decisions are used to adjust weight combinations during every iteration of the simulation. Such an algorithm obviously has some limitations because a lot of possible adjustments to the weights can not be tested/explored. Moreover, we only use four features in the handcrafted agent, which may not be sufficiently representative. As a result, we need an approach that takes more underlying features into consideration, together with a more systematical way to improve weight combinations other than just by intuition. With these goals bearing in mind, we propose a second approach that utilized the genetic algorithm, a powerful local search algorithm.

[1]Note that this is special compared with other features because we want it to be as large as possible, so we will assign a negative weight to it.

In the genetic algorithm approach, we would use a feature set that includes a total of 10 features. These features are:

1-4. features same as Handcrafted agent

5. number of blocks currently on the board
6. number of rows with at least a hole in it
7. max height difference between the lowest column and highest column
8. number of cells that is not filled with no blocks above it and two horizontal neighbors filled
9. the height where the highest hole is located
10. number of occupied rows which is above the highest hole such that if removed, the highest hole is no longer a hole

Now that once we have our feature set, we would construct our genetic algorithm approach start by defining the following terms:

- **Chromosome**: An array of 10 numbers representing our weights combination, each weight ranges from -1 to 1.
- **Fitness**: Number of lines cleared before the game ends.
- **Selection**: After ranking each state based on fitness score, we will remove p percent of states with low fitness scores and use remaining states as parents for next generation.
- **Crossover**: Take a combination of weights among the remaining parents and make it the chromosome for the children (detailed implementation is shown below).
- **Mutation**: After crossover, one random weight value in the chromosome of the child state will be mutated with a certain probability q.

The game would start by generating random chromosomes for M children, and for each of these children, we will run N simulations on it. The average lines cleaned will be recorded for each child and will be used as the fitness score for them. Next, we would begin our selection process, which would remove $p\%$ children and promote the remaining children to be parents. For this group of parents, we would randomly choose M pairs of parents and perform crossover and mutation operations to get M children for the next generation. Then, for each child in this generation, we would start the above the process all over again. That is, we would run simulation, record fitness, remove least fit offspring, and generate successors. We would stop the game until there is no significant improvement in performance between the parents and the children. Finally, we will choose the child with the highest fitness score and use its chromosome as our final weight combination.

One important thing worth noting is the implementation of crossover. When determining the exact combination of chromosome from the parent states, we will consider the fitness scores of parents because if one parent has a higher fitness score than the other, the former chromosome should have been more optimal than the latter one. Therefore, we would want the child to "inherit" more from the first parent. Let's assume the first parent's fitness score is $f1$ and that of second parent is $f2$. Without loss of generality, assume $f1 \geq f2$, then we define the combination as follows:

- if $\frac{f1}{f2} = 1$, we will randomly take 5 weights from the first parent and fill the remaining weights from the second parent.
- if $0 < \frac{f1}{f2} - 1 \leq 0.25$, we will do a 6:4 sampling.
- if $0.25 < \frac{f1}{f2} - 1 \leq 0.5$, we will do a 7:3 sampling.
- if $0.5 < \frac{f1}{f2} - 1 \leq 0.75$, we will do a 8:2 sampling.
- if $0.75 < \frac{f1}{f2} - 1$, we will do a 9:1 sampling.

Notice that this scheme involves many hyperparameters, such as the lower and upper bounds of the ratio, and in theory, we should determine this during our training. However, doing so will introduce too many degrees of freedom during training, and to avoid this, we will insist on this scheme.

In conclusion, in the genetic algorithm approach, we eliminate a lot of human assumptions. That is, what weight value should be changed, and by how much. We would also take into consideration more features that may further eliminate human assumptions of which feature could have an underlying impact on the game-play or not. Despite the amount of computation with this approach, we are more likely to find an optimal weight combination compared to a handcrafted agent.

Reinforcement Learning Agent

The genetic algorithm mentioned above is intelligent in the sense that it will automatically find optimal weight combinations. However, notice that it is still not perfect because the state features are always chosen by humans, and they are not necessarily representative. More concretely, we may lose some information since we will represent all the successors' states, which is of size 20×10, using only ten features. Thus, we will develop a new algorithm that eliminates human-chosen features and finds a feasible representation itself.

As a result, we naturally lean towards the fields of reinforcement learning and will start with the Q-learning introduced by Chris Watkins in 1989. The detailed deduction and convergence proof are pretty complicated and are out of the scope of this report. However, the final results of Q-learning are incredibly similar to dynamic programming, where we maintain a Q-table with current states s_i as column and available actions a_j as rows (Table 1).

	a_0	a_1	$\cdots$	a_n
s_0	0.91	0.65	$\cdots$	0.45
s_1	0.86	0.14	$\cdots$	0.19
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
s_m	0.26	0.61	$\cdots$	0.27

Table 1: Q-table example with random initialization

And for each step, we will update the entry (s_t, a_t) using the formula

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_a Q(s_{t+1}, a)\right)$$

where we have two hyper-parameters embedded in this formula:

- α: learning rate, which determines to what extent newly acquired information overrides old information.
- γ: discount factor, which determines the importance of future rewards.[2]

On top of that, we also need to manually determine three important setups:

- s_i: all the possible states for the environment represented by a 20×10 bitmap.
- a_j: all the possible actions a player can take for a state.
- r_t: reward received when moving from the state s_t to s_{t+1} and is determined by the reward function.

With all these hyper-parameters and setups, we will let the agent interact with the environment on its own. At each state, the agent will always choose to perform the action with the highest Q-value and update the Q-table according to the updating formula.[3] We will stop Q-learning until the agent completes the game, and repeat this process for large enough epochs so that the Q-table converges.

Now, we will begin with a naive adaptation to the Tetris by defining states to be all the possible game states for a Tetris board of size 20×10 and defining actions to be the set $\{clockwise\ rotation, counter\text{-}clockwise\ rotation, left, right, down\}$.

Theoretically, we could tackle this Tetris problem directly using the Q-learning setups mentioned above, but we are facing two severe hindrances:

1. The reward is very sparse because all the actions except *down* are perfectly reversible. For example, $left$ can be reversed by $right$, and $clockwise\ rotation$ can be reversed by $counter\text{-}clockwise\ rotation$. As a result, in the exploration stage, it will take the agent a long time before it can achieve a reward[4], which is particularly bad for training purposes.
2. Furthermore, according to Phon-Amnuaisuk's paper, the number of stable states for a 20×10 Tetris board is in the order of 10^{60} (Phon-Amnuaisuk 2015), which means our Q-table will be of size approximately 5×10^{60}. This table is way too large to store in RAM and the training time for such a large table is also unbearable.

[2]The term $\max_a Q(s_{t+1}, a)$ represents the estimation of the optimal future value.

[3]Here, it is not always true that the agent will perform the action with the highest Q-value. Some techniques like ϵ-greedy will choose the optimal action with probability ϵ and a random walk otherwise. We will talk more about this later in this section.

[4]Notice that according to the Table 3, we will only get a reward when we place a Tetromino so we won't get anything during the placement process.

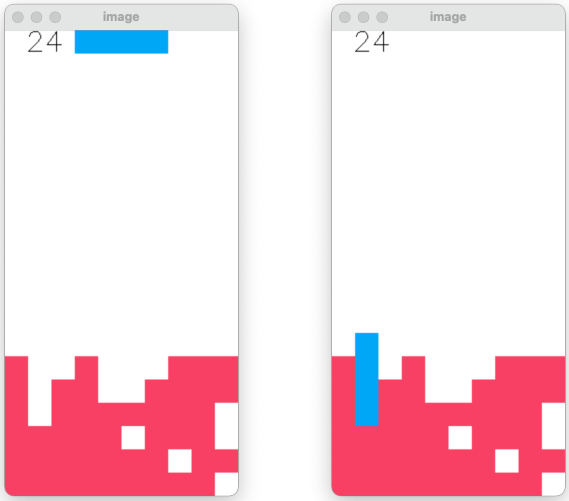


Figure 1: Grouped action example

Now, we will conquer these two hindrances and develop a practical architecture for this problem. For the first one, inspired by Stevens's idea of *grouped action* (Stevens 2016), we can leverage the successor function from the genetic algorithm and reduce the dimension of Q-table. More precisely, instead of defining action a_i to be one of the 5 single moves, we will use a *grouped action* to represents a sequence of moves until the current Tetromino is landing.[5] For example, if we got an L-shape Tetromino as shown in Figure 1, one potential *grouped action* will be $\{left \times 3,\ clockwise\ rotation \times 1,\ down \times 10\}$, which is essentially the successor function used in the genetic algorithm.

With the new definition of *grouped action*, we can reconstruct our Q-table and rewrite our updating formula. Notice that because after each grouped action, it's guaranteed to be a stable state (i.e., there is no floating Tetromino), we can simplify the two-dimensional Q-table (Table 1) into one-dimensional (Table 2).

s_0	s_1	$\cdots$	s_{m-1}	s_m
0.91	0.65	$\cdots$	0.45	0.19

Table 2: New Q-table with random initialization

with the modified updating formula:

$$Q(s_t) \leftarrow (1-\alpha)Q(s_t)+\alpha\left(r_t+\gamma \max_{s_i \in successor(s_t)} Q(s_i)\right)$$

With the reconstructed Q-table and modified updating formula, now for each state, the agent will choose the successor with the highest Q-value instead of a specific action. This successfully solves the sparse reward problem because now, after each grouped action, we will always get a non-zero reward. However, this setup inherits the drawback of requiring enormous storage space and training time because the number of states remains 10^{60}.

To overcome the second hindrance, we need to understand why we have so many states in our Q-table. The problem lies in the fact that we have 20×10 cells in the game board, and the number of possible states is exponential in the number of cells. This situation reminds us of how we encode images during image classification, and naturally, we will consider using Convolutional Neural Networks (CNN) to encoding our state (i.e., a 20×10 bitmap). This idea is first introduced by Google DeepMind in the paper Playing Atari with Deep Reinforcement Learning (Mnih et al. 2013), and we will adapt this idea to Tetris to come up with the architecture as shown in Figure 11 in **Appendix A**.

This Deep Q-network (DQN) will take a 20×10 bitmap as input and output a non-negative real number as the score for that bitmap. This behavior is essentially the same as what we did in Q-learning except that the DNQ will generate the score on the fly while Q-learning will merely look up the score. At its core, the DQN is trying to approximate the Q-table using a reasonable amount of time and space.

Compared with other CNN architecture, our DQN is relatively small and straightforward, mainly because of the tiny input size. A detailed summary and visualization of the network can be found in **Appendix A**. On top of that, the pseudo-code to train the DQN is shown in Figure 2.

Algorithm 1: Pseudo-code for Training

Function `train`(cur, DQN)**:**
  Input: cur is current game state and DQN is the network, which supports two operations $predict$ and fit
  Output: nex is the next game state
  $states \leftarrow successor(cur)$
  $maxQ \leftarrow 0, reward \leftarrow 0, maxState \leftarrow$ **None**
  for $state$ ***in*** $states$ **do**
    $score \leftarrow DQN.predict(state)$
    if $score > maxQ$ **then**
      $maxQ \leftarrow score$
      $reward \leftarrow reward(state)$
      $maxState \leftarrow state$
    end
  end
  $Q' \leftarrow (1-\alpha)\cdot maxQ+\alpha\cdot(reward+\gamma \cdot maxQ)$
  $DQN.fit(maxState, Q')$
  return maxState

Figure 2: Pseudo-code for Training

Now, there are two details worth mentioning during the training process.

1. **ϵ-greedy Algorithm**

   The tradeoff between exploration and exploitation is one of the most critical challenges for most of the reinforcement learning problem, and our DQN is no exception.

[5] In other words, each *grouped action* refers to a final placement for a specific Tetromino.

Note that for the current setups, the only chance for our network to explore comes from the random initialization of the system, and once it finds a solution, exploitation will dominate exploration because we always choose the successor with the highest score. This is not ideal because we may miss some potential shortcuts[6], and as a result, we will apply the ϵ-greedy algorithm during the training phase. In brief, it will stick to the optimal successor with a decaying probability ϵ and randomly choose a successor otherwise (Tokic 2010). Thus, we will update the training algorithm in Figure 3.

Algorithm 2: Training with ϵ-greedy

```
Function train(cur, DQN):
    states ← successor(cur)
    maxQ ← 0, reward ← 0, maxState ← None
    for state in states do
        score ← DQN.predict(state)
        if score > maxQ then
            maxQ ← score
            reward ← reward(state)
            maxState ← state
        end
    end

    // ϵ-greedy algorithm
    p ← uniform(0, 1)
    if p > ϵ then
        maxState ← random(states)
        maxQ ← DQN.predict(maxState)
        reward ← reward(maxState)
    end
    ϵ ← ϵ − δ                    // ϵ decaying

    Q' ← (1−α)·maxQ+α·(reward + γ · maxQ)
    DQN.fit(maxState, Q')
    return maxState
```

Figure 3: Pseudo-code for Training with ϵ-greedy

Note that we need a decaying ϵ because we want to explore more in the beginning, and as we approach the end of the training phase, we want to exploit more.[7]

2. **Training by Part**

According to the experiment results from Steven's paper (Stevens 2016), another challenge we will face during the training is that each game will last for a long time, mainly because of the large 20×10 game board. Therefore, we will solve this problem by training our network part by part, where we will start with a small board and gradually increase the board size. A more detailed implementation about this can be found in **Experiment Design** subsection under **Results** section because it involves choosing a lot of hyperparameters.

In summary, DQN architecture successfully solves the problems of sparse rewards and enormous states, which gives us an efficient and effective estimation for the original Q-learning in Table 1. Moreover, with CNN's powerful encoding capability, we don't need to choose the features manually, and the architecture will automatically find the optimal features during backpropagation. As a result, we can say that DQN architecture has the potential to outperform the handcrafted and local search agents.

Results[8]

Environmental Setup

For the implementation of the game Tetris, we referenced the Tetris-AI implementation (Faria 2019). The board is 20×10, and each Tetromino is a list of four positions that it occupies. At every step, it will generate the appearance of the next Tetrominos at random. The state will be defined as a 20×10 matrix, where each entry could be one of 0, 1, or 2, represents the empty block, the occupied block, and the block occupied by the current Tetromino, respectively. For each state, all the features needed by handcrafted and local search agents will be collected, and we will use them to calculate the heuristic value.

For the successor function, given the current board state and the next Tetromino, it will try all the possible locations and rotations that the Tetromino could land. Starting from the top left corner, the successor function will try all the rotations (1, 2, or 4 rotations depending on the shape of Tetromino. 1 rotation for O shape, 2 rotations for I, and 4 rotations for other shapes). Then shift the Tetromino 1 unit to the right and repeat the process again until we reach the right top corner of the board. That is when we can figure out all the possible moves, and according to the type of agents, the successor function will return different results. More precisely,

- For the handcrafted agent, we will return a list of values for 4 feature
- For the local search agent, we will return a list of values for 10 feature
- For the reinforcement learning agent, we will return a list of 20×10 bitmaps representing all the successor states

Finally, the agent will return its decision to the environment, which will update the current state correspondingly.

Evaluation Metrics

According to previous researches and the game mechanism, we have two reasonable metrics to choose from. One is to count the total number of Tetromino dropped before the game ends. And the other is to count the number of lines cleared before the game ends. They both sound reasonable, but we would like to use the second metric for our evaluation. The number of Tetromino dropped somehow describes "how long" does the game played, but this does not indicate

[6] In other words, the selected successor may not be optimal for long-term.

[7] This gives two extra hyperparameters ϵ and δ that need to be adjusted during training.

[8] Here is the github link to the implementation: `https://github.com/shuheng-cao/CS486GroupProject`

"how good" the model is. It is still possible to fill the board without clear a single line, and it is also possible to clear several lines with very few Tetromino dropped. In order to make it live longer, the purpose is to remove lines whenever it could and clear lines as much as possible. So we pick the number of lines cleared as our evaluation metrics. In general, the score of the game is highly related to the number of lines cleared. The more lines cleared, the more score we will get finally, so it means the model plays "better". On the other hand, according to the previous researches on Tetris agent (Algorta and Simsek 2019), the rules and the way that the score is rewarded can be different. We finally want to compare our model with the agents that already exist. In order to make a fair comparison, all of the agents will use the same evaluation metric, which is the number of lines cleared before a game ends so that we could tell which model has a better performance in the game.

One thing worth mentioning about the evaluation metric is that we will simoutanously use two metrics when training the Reinforcement Learning agent. The first metric is the one mentioned above, and it will be used as the primary standard to compare the performance of different hyperparameter combinations during validation. On top of that, this metric will also be used to compare with other algorithms mentioned in this paper as well as results from other papers. However, one problem with the primary metric is that its rewards are very sparse, where the agent needs to wait for several Tetrominos before it could clean a line and achieve one mark. As mentioned before, this problem is particularly bad for training a DQN because, without frequent rewards, it is difficult for the network to learn anything from backpropagation. Therefore, we decided to reshape the primary metric into the following table.

Situation	Reward
Game over	-100
Clean up k lines	$10 \times k^2$
Safely landing a piece	1

Table 3: Reward Function

Note that by design, the new metric is positively correlated with the primary metric in the sense that both of them encourage cleaning as many lines as possible. However, the new metric will also introduce a relatively small reward (compared with cleaning a line) for safely landing a Tetromino for the sake of more frequent stimulation for backpropagation. Meanwhile, this metric also introduces a substantial penalty for losing the game to produce strong negative feedbacks for the network.

In summary, we will use the total number of lines cleaned as our primary metric to compare different algorithms while using a specially-designed metric to train the reinforcement learning agent.

Experimental Designs

As mentioned above, since we implemented a Tetris environment, we can say that we have unlimited training data. Therefore, we can train our algorithm as many games as we want as long as time permits.

- **Handcrafted Agent**

  The experimental design for the handcrafted agent is relatively straight-forward. Since we are merely relying on human observation and intuition, we need to keep track of each game manually. Therefore, we will use a training set of 3 games, a validation set of 100 games, and a test set of another 100 games. The training process is rather simple. We will watch the agent playing for three rounds (training) and let it play for itself for another 100 rounds (validation). Meanwhile, we will analyze its mistakes during training and calculate the average score for the validation. Then, according to the analysis, we will adjust the weights correspondingly, and if the combination of the new weights yields a better validation score, we will use it. Otherwise, we will try some other updates, and if we can't improve the validation result anymore, we will stop. Finally, we will use the best weights combination as our final result and test it on another 100 games.

  As for the weight initialization, we will start with the weights $(-1, 3, 2, 4)$ with the following reasoning. Ultimately, the higher our board is, the less chance there is for the game to last longer. So, we intuitively assign the most weight to the $4th$ feature (maximum height). Now we want to assign weights to the $3rd$ and the $2nd$ feature. We notice that holes make it hard to be cleared; and it may consequently lead to accumulated height of the board. Therefore, in our initialization, we want to assign a relatively higher weight to the second feature, which is the number of holes. Finally, as mentioned in Methodology part, we would like the weight for the first feature to be negative because we want the number of lines cleared to be as large as possible. Hence, we would choose the number -1 as the initial weight to the $1st$ feature.

- **Local Search Agent**

  Again, we will decide on the train-validation split first. Since we don't need to manipulate the weights manually for the local search agent, we will train the agent on more games. We are going to train the agent for 1,000 games[9], validate on 100 games, and eventually test on another 100 games.

  As mentioned in the Methodology section, the experiment would have three main hyperparameters, namely selection probability p, mutation probability q, and the number of children M. Since the number of hyperparameters are tolerable, we will do a complete hyperparameter search on these values:

  - **Selection probability**: $p \in \{0.4, 0.5, 0.6\}$
  - **Mutation probability**: $q \in \{0.01, 0.1\}$
  - **Number of children**: $M \in \{50, 100\}$

  More precisely, we will train on $3 \times 2 \times 2 = 12$ different combinations of hyperparameters and find the one with highest validation score. After this complete search, we will perform some further explorations on a more precise

[9] This is essentially the N we mentioned in Methodology.

level. For example, if we found $p = 0.5, q = 0.1, M = 100$ to be the best-performed combination of hyperparameters, we will do another complete search on

- **Selection probability**: $p \in \{0.45, 0.5, 0.5\}$
- **Mutation probability**: $q \in \{0.08, 0.12\}$
- **Number of children**: $M \in \{80, 120\}$

We may repeat this "zoom in" search several times if time permits and at last, we will test the best model on another 100 games to compare it with benchmark as well as the reinforcement learning agent.

In my opinion, the rationale for the complete search is that we don't have a proper initialization to start on, so we have no choice but to begin with a broad range of hyperparameters. After the first exploration, we will get some ideas about the optimal hyperparameters, and we could aim towards a more precise decision. This is a time-efficient way of finding the optimal hyperparameters when we don't have any prior knowledge.

- **Reinforcement Learning Agent**

  Similar to the train-test split mentioned in genetic algorithms, for each hyperparameter combination, we will train the network for 1,000 games and validate for 100 games, where we will measure both metrics (primary metric and reward) during training but only calculate the primary metric for validation. After we find the optimal hyperparameters, we will test the final architecture for 100 games and use this result to compare with others. As we mentioned in the **Methodology** section, we will only perform the ϵ-greedy algorithm and training by part techniques during the training process because we don't want to introduce any randomness to the validation or testing.

  As for the hyperparameter, we will leverage results from Stevens's paper (Stevens 2016) and initialize our hyperparameters using their suggestions as follow.

  - **Optimizer**: $Adam$
  - **Regularization**: Batch normalization and dropout with a retention probability of 0.75
  - **Learning rate**: $\alpha = 2 \times 10^{-6}$
  - **Discount rate**: $\gamma = 0.9$
  - **Randomness**: $\epsilon = 0.95$ and $\delta = 0.0005$

  However, since our reward function, DQN architecture, and environment implementation are different from theirs, we need to adjust our hyperparameters according to the validation results, where we need to explore the optimal combination of hyperparameters similar to what we will do for the Genetic Algorithm mentioned above.

  Now, recall that we introduced the special technique *training by part* technique in the **Methodology** section and here is the detailed explanation:

  We will decompose the training process into three parts for the sake of time efficiency as follow:

  - Firstly, we will train the DQN on a 5×10 game board for 400 epochs.
  - Then, we will train it on a 10×10 for another 400 epochs.
  - Finally, we will train it on the full board (20×10) for 200 epochs.

  This will give us a total of 1,000 training epochs, but this design should be much faster than trained directly on the full board because the time needed for each game is shortened. Technically speaking, the *training by part* also involves some hyperparameters, such as size of the game board and length for each sub-training process. However, we don't want to introduce too many degrees of freedom to the hyperparameters so we will make the assertion that our choice is optimal.

  At last, a detailed explanation on implementation of *training by part* can be found in **Appendix B**.

  In conclusion, the experimental design for the Reinforcement Learning agent consists of three steps of training where we will employ ϵ-greedy algorithm and partial weight initialization for the sake of efficiency and effectiveness.

Implementation

As mentioned in the environment setup and design subsections, the implementation of this report mainly involves three parts: a Tetris environment to interact with ML agents, the local search agent, and the Reinforcement Learning (RL) agent. More elaborate explanations for all these three parts are as follows:

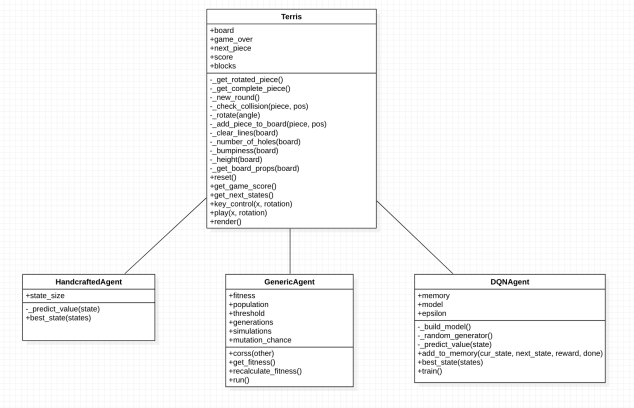


Figure 4: UML

- **Tetris Interface**

  The implementation for the Tetris interface is very important. According to the design and the UML showed in Figure 4, the methods of Tetris interface will be invoked by all three models. Since the three models will use the methods from Tetris class such as the successor function (get_next_states()), the feature value calculation functions, and play a Tetromino by placing it into the current board and the board update function is also needed. These methods will be frequently used during the training and validation process. Due to the large number of games need to train the model, the efficiency of Tetris interface implementation will result in a significant effect on the models' efficiency and the time cost that we need to spend.

Several optimizations during the implementation of Tetris interface have been made. For the feature value calculation functions, since multiple features are needed to evaluate, and most features need to go through the entire board. It is very inefficient if we use separate methods for every one of features. It will result in many duplicate calculations. A potential solution is to combine several feature value calculation functions together, for each time we go through the entire board, return as much feature values as possible. For example, in order to figure out the number of holes, it is necessary to check every individual cells. At the same time, we could also record the highest hole position, number of rows with holes, well cells, column transitions and row transitions. To calculate the maximum height, we need to go through each columns, and at the same time, we could store the information of the height of each columns, and it will be easier to update the corresponding height in the future. This height function would return the maximum height, minimum height, sum of height, bumpiness at the same time.

For the successor function, instead of return the entire board for each possible successor states, we make it return the bitmap, with is a huge support for reinforcement learning agent. Also, it contains the information of next step's horizontal shifts, the angle of rotations and the feature values. So that the agent could use feature values to decide the optimal solution and whenever a decision is made, we could make a play by using the horizontal shifts and rotation, which would improve the time complexity and extra space usage. Also, the successor function needs to check every single possible moves. Using a brute force algorithm until a collision occurs. However, the part of the board which is beyond the maximum height is the same as an empty board. It is impossible to have a collision beyond the maximum height. We have the maximum height calculated and stored during the feature value calculations, thus exploring the area beyond the maximum height is a waste of time. Thus we start and check collisions from the level of maximum height. Since one of our purpose is to reduce and keep the maximum height as low as possible, and we do not want the maximum height to grow fast. Thus by applying this approach, we could improve the efficiency significantly.

- **Local Search Agent**

  For the implementation of the local search agent, one important step is to calculate the features' value of the current board after a tetromino drops. As introduced in methodology part, we have 10 features for the local search version. An implementation challenge would be to compute the values for all features with least computation time. In order to optimize it, instead of having a helper function/method for each feature, we tried to compute as many features as we can inside one function call. This way, we would have least amount of function calls for computing the feature, thereby eliminating all unnecessary loops inside a function.

  Another technical difficulty involves frequently copying the game state (current board holing the tetrominos dropped so far). In order to drop a most desirable tetromino, we have to try all possible dropping locations. This involves copying the board over and over again. To improve performance, we would do a shallow copy of the current board every time.

  Aside from the aforementioned two aspects of the implementation, the implementation of genetic algorithm was straightforward in terms of selection, crossover, and mutation process.

- **Reinforcement Learning Agent**

  During the implementation of the RL agent, we noticed two significant challenges to the efficiency of the training process:

  – As mentioned in the implementation details above, when we first build the Tetris environment, it aims to optimize the performance of calculating the features for the local search agent. However, since the only input for the RL agent is the bitmap representation of the game board, we waste a lot of computation power on the unnecessary features, especially for the function that generates the successor states.

  – Moreover, as shown in **Algorithm 2** in Figure 3, the training process involves two stages, calculating successors and fitting the model. This algorithm will work, but a crucial drawback arises as we need to go back and forth between the CPU and GPU frequently. More precisely, the environment (which relies on the CPU) must wait for the agent (which depends on the GPU) to fit the data while the agent must wait for the environment to generates successors. This mutually dependent situation is notoriously bad for training efficiency.

  Now, in order to optimize the training efficiency for Reinforcement Learning, we overcome the two challenges using the following techniques.

  – Since the current Tetris environment is closely bound up with the local search agent, it is impractical to use it for the RL agent. As a result, the RL agent uses an adapted version of Tetris where we eliminate all the feature calculations, and the changes can be summarized as the updated UML below:

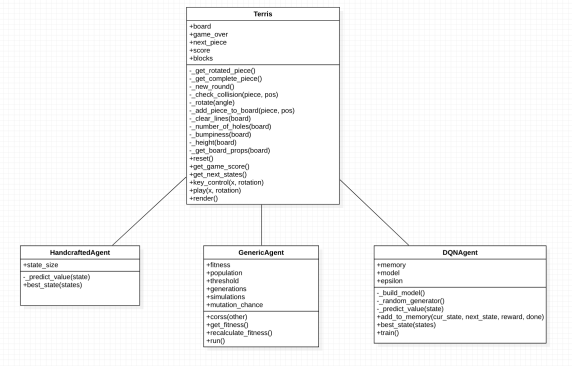


Figure 5: UML

– On the other hand, to solve the frequent switch between CPU and GPU, we take advantage of the ideas from mini-batch gradient descent and dynamic programming. More concretely, instead of fitting the data right after its generation, we will push the $(state, Q-value)$ pair into a queue and only train on the data after a game finishes. Notice that such a change will not improve training efficiency but also provide reusability of training data if the queue is large enough.

In summary, with these two changes, we reduce the time needed to train for 200 steps from 33 seconds to 19 seconds on a machine with two Intel Xeon E5-2660 CPUs and one Tesla T4 GPU.

Experimental Performance

First and foremost, Table 4 is a compact summary of the best performance (trained using the optimal hyperparameters) of all three agents mentioned in this paper:

Agents	min steps	avg steps	max steps
Handcrafted	23	210	2072
Local Search	20176	40138	52364
RL	102	339	1078

Table 4: Summary for Three Agents

For the experiment performance subsection, we will concentrate on the local search and reinforcement learning agent because there is not much to improve in the handcrafted agent, as mentioned in the experiment design subsection.

- **Local Search Agent**

  Overall the performance of local search agent approach is quite satisfying. The best weight combination trained by the genetic algorithm yielded an average score of more than 40000 and a maximum score of more than 50000, which is dramatically more successful than a Reinforcement Learning agent approach.

  During the training process, we aimed to train three hyperparameters: selection probability p, mutation probability q, and number of children in the initial generation M. For each child in any generation, we would perform 1000 games on it and use the mean score (out of a total of 1000 games) as the score for a child (a given chromosome). After the testing for one possible hyper-parameter combination finishes, we would have our fittest member. Then we would use the weight combination associated with the fittest member to perform another 100 games and record the minimum score, the mean score and the maximum score. The overall training performance is illustrated in Figure 4. As we can see, the fittest member has a chromosome of $p = 0.6, q = 0.1, M = 100$.

  After we get the fittest chromosome resulted from training, we would then perform validation process in a way that is specified in Methodology part. That is, we would do a complete search against: $p \in \{0.4, 0.5, 0.6\}$, $q \in \{0.01, 0.1\}$, and $M \in \{50, 100\}$ In this step, we recorded the minimum, mean and maximum validation score for each chromosome, as shown in Figure 5. It is shown that the optimal hyper-parameters resulted from the validation process is $p = 0.55, q = 0.08, M = 120$.

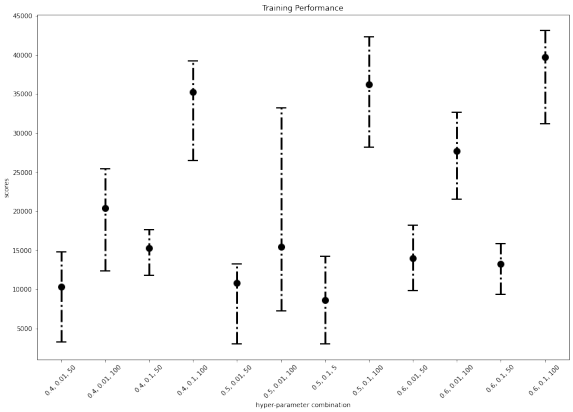


Figure 6: Training Process

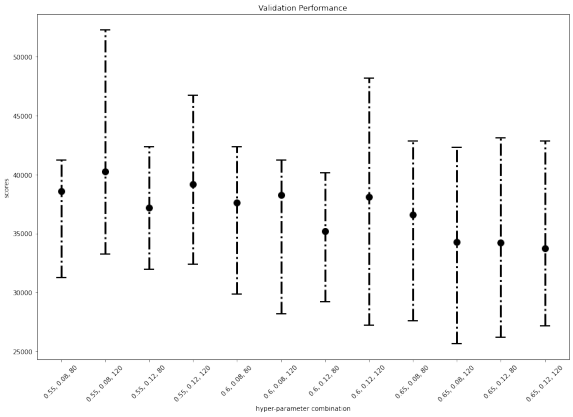


Figure 7: Validation Process

An interesting discovery from the data exploration is that the average performance is positively related to the number of children in the initial generation. In other words, if p and q remain the same, then the more M is, the higher the game score is. It makes sense because because in each generation we have more members to perform crossover, which probabilistically increases the chance of generating a a fitter child.

The biggest challenge for the performance of implementation for local search agent is with our algorithm's runtime. The whole process including training, validation, and testing took more than a week to finish. However, given the sheer amount of the data and computing tasks (a substantial amount of copying), we would argue that the long run-time is inevitable.

- **Reinforcement Learning Agent**

  According to the RL agent's design and structure, we can see that one significant hindrance for training the agent

is that we need to decide the value of at least 6 hyperparameters as list below.

- Whether we should use batch normalization (BN).
- Whether we should use dropout (DO).
- The value of the learning rate (α).
- The value of the discount rate (γ).
- The value of the randomness rate (ϵ).
- The value of the randomness decay (δ).

To make things even worse, our previous adaptation for efficiency introduces two more hyperparameters, the batch size (BS) and the queue size, which is also known as memory size (MS). Therefore, technically speaking, we need to adjust and optimize a combination of 8 hyperparameters, which is nearly impossible to achieve through a full grid search. Fortunately, as mentioned in the Experimental Design subsection, there is some previous work in solving Tetris by RL, providing valuable information about hyperparameters. As a result, instead of exploring all the combinations of hyperparameters, we start with the recommended choices (baseline) and only alter one hyperparameter at a time while keeping all other fixed. This schedule significantly decreases the number of experiments needed, from exponential to linear, and Figure 8 shows the visualization of the performance for altering the first 6 hyperparameters. Notice that the black points in the middle represent the average steps for a specific combination of hyperparameters, while the lower and upper bound of the dashed line stands for the minimum and maximum steps for that combination.

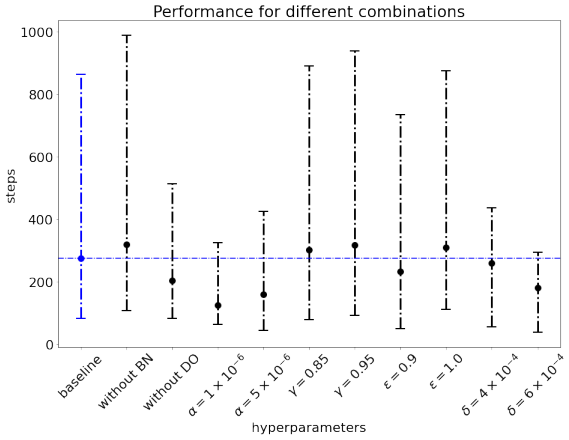


Figure 8: hyperparameters performance comparison

On top of that, since the previous work didn't apply any optimization on the training algorithm in Figure 3, they are using the default value for the batch size and memory size, i.e., $BS = 1, MS =$ length of game. Therefore, for these two hyperparameters, we need more exploration than the previous ones, and the performance results are shown in Figure 9.[10]

[10]The tabular data can be found in **Appendix C**

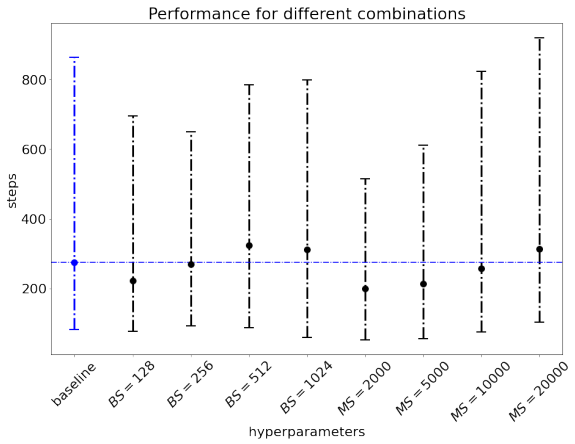


Figure 9: hyperparameters performance comparison

Comparing each combination's performance with the baseline, we expect the following hyperparameters to be an optimal combination.

$$BN = \text{false}, DO = \text{true}, \alpha = 2 \times 10^{-6}, \gamma = 0.95$$
$$\epsilon = 1.0, \delta = 5 \times 10^{-4}, BS = 512, MS = 20,000$$

Notice that the most significant improvement comes from the removal of the batch normalization layer (which results in a 14% increase in average steps) and a large memory size (which results in a 12% increase in average steps). From the authors' perspective, the intuitions behind these trends are as follows:

- Firstly, since we are using a combination of ReLU and leaky ReLU as activation functions for the whole network, we don't necessarily need normalization since concentration on zero doesn't help much on ReLU. Therefore, removing batch normalization will help in reducing the complexity and achieving better results.
- On top of that, using a large memory size will help us remember more $(state, Q - value)$ pairs, which is equivalent to a larger dataset. As a result, this should also help in improving performance.

Now, with these hyperparameters, we indeed achieved a better validation result, as shown in Table 4, and the training curve in Figure 10 also gives us some sense of why the combinations make sense. For example, a large randomization rate will dominate the early training period, so we have almost no improvement in the first half. But this is beneficial for the long term as we are exploring more and the latter half is improving a lot.

Finally, with a well-trained RL agent, we could have a deeper understanding of how deep Q-learning (DQN) works by associating visual states with its q-values. A few examples can be found in **Appendix D**.

Reflection

As we can see in the two subsections above, our implementation is quite comprehensive in the sense that it involves numerous object-oriented programming techniques as well as

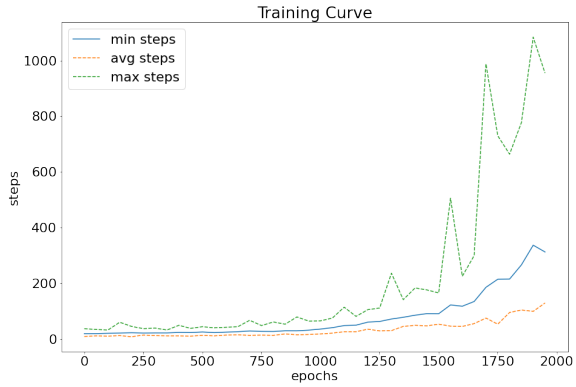


Figure 10: training curve for RL agent

some of the cutting-edge machine learning methods. Consequently, the authors will note some lessons learned during the implementation processes, and again, it naturally falls into three parts.

- **Tetris Interface**

  The lesson that we learned from implementing Tetris interface is the design. Since the Tetris interface cooperate with all three agents, thus the key point is to design a barrier-free implementation, so that agents would have enough flexibility to get every information they need from the Tetris interface. Design and implement Object Oriented Programming in python is a new experience for us. We need to learn how to think about the whole structure at a very high level before we start. We need to figure out the API that we might need for each agent. This is a challenging part since we would like to have the same API work for all agents and also all sufficient information is provided. We also learned working as a team. We need to highly cooperate with each other so that is will be easier for us to combine separate parts together, which involves a lot of pre-planning and team communications.

  On the other hand, the optimization is another challenge. As mentioned before, the performance and efficiency of the Tetris interface significantly affects the time cost of other agents since agents relies on the APIs of the Tetris interface. At the beginning, the efficiency is not quite satisfiable. It needs relatively long time to get all successors and feature values. We spend a lot of time thinking about how to improve the efficiency and remove duplicate calculations without reducing the consistency. It is a valuable lesson about optimizing implementation performance.

  Last but not the least, another thing that we learned is visualization of Tetris with python. And we also implemented keyboard control function into the Tetris. It is also a playable Tetris game itself. Visualization helps to debug such as checking whether we are getting the right feature value for the agents. Also, for some extreme cases, instead of using random ordered Tetrominos, we would like to construct a state by our own to test certain extreme performance. Thus the Tetris interface could take inputs from the user other than control the movement of Tetromino, but also allow the user to force the type of next Tetromino. We learned the usage of cv2 pakage in python.

- **Local Search Agent**

  For the genetic algorithm, I've learned two important lessons.

  First, it really did the good job searching for the best chromosome. Without explicitly writing technical details to play the game well. We simply told the algorithm what is considered to be optimal and what is not. The actual result is much more desirable than using handcrafted agent and reinforcement learning agent.

  Second, although the algorithm did a good job calculating the weight for each feature, that does not mean that it can find all underlying features that would affect the gaming score. After all, we only took into consideration of 10 features for this local search implementation. Maybe there are other underlying features we fail to identify. Therefore, to improve search performance, we could add as many features as possible to our chromosome.

- **Reinforcement Learning Agent**

  In the authors' opinions, the most valuable lessons learned from the analysis and implementation process are that theory doesn't always correspond to the practice. In this report, we analyze how and why the reinforcement learning agent should surpass the local search agent, but after we implement both, it is later that overshadows the former. The underlying reasons will be analyzed in greater detail in the Discussion section, but in brief, the lesson is that the most complex model isn't necessarily the best model. On top of that, the authors also learned many practical skills during the learning and implementation process. Firstly, none of the authors had any preknowledge about q-learning, so we need to study everything from scratch by ourselves. This learning process should be beneficial for further career, either in academics or industry. Moreover, during the implementation process, the authors encountered plenty of challenges that will never show up in the homework for example, spending days training and validating to find optimal hyperparameters, spending hours dealing with the frequent switch between CPU and GPU, etc. These experiences give us a more in-depth understanding of Reinforcement Learning as well as practice our problem-solving skills in real life.

References

Algorta, S., and Simsek, O. 2019. The game of tetris in machine learning.

Bertsekas, D. P., and Tsitsiklis, J. N. 1996. *Neuro-Dynamic Programming*. Athena Scientific, 1st edition.

Burgiel, H. 1997. How to lose at tetris. *The Mathematical Gazette* 81(491):194200.

Faria, N. 2019. tetris-ai.

Gärtner, T.; Driessens, K.; and Ramon, J. 2003. Graph kernels and gaussian processes for relational reinforcement learning. In Horváth, T., and Yamamoto, A., eds., *Inductive Logic Programming*, 146–163. Berlin, Heidelberg: Springer Berlin Heidelberg.

Matt Stevens, S. P. 2016. Playing tetris with deep reinforcement learning. *stanford.edu*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing atari with deep reinforcement learning.

Phon-Amnuaisuk, S. 2015. Evolving and discovering tetris gameplay strategies. *Procedia Computer Science* 60:458–467.

Stevens, M. 2016. Playing tetris with deep reinforcement learning.

Tokic, M. 2010. Adaptive -greedy exploration in reinforcement learning based on value differences. 203–210.

Tsitsiklis, J. N., and Roy, B. V. 1996. Feature-based methods for large scale dynamic programming.

Wei-Tze Tsai, Chi-Hsien Yen, W.-C. M., and Yu, T.-L. 2013. Tetris artificial intelligence.

Appendix A

The structure of the network is shown in the Figure 11 and here are the detailed information about each layer:

- **conv1**: a convolutional layer with filter size 3×3, output channel size 32, and leaky ReLU activation function.
- **pool1**: a max-pooling layer with filter size 2×2.
- **conv2**: an identical layer as **conv1** except that the output channel size to be 64.
- **pool2**: an identical layer as **pool2**.
- **flatten:** this is simply a flattened version of the **pool2** feature map.
- **fc**: this is a fully connected layer of size 256 with ReLU activation function.
- **score**: this is a fully connected layer of size 1 with ReLU activation function.

Appendix B

The sequential training process of *training by part* is actually not as easy as it sounds like, and the main problem here is the difference in length of the flattened vectors:

Input Size	Flattened Vector Size
5×10	512
10×10	1024
20×10	2048

Table 5: Flattened vector sizes for different input

The difference between these vectors' sizes will mess up the weight matrix of DQN architecture, so we need to initialize the weight matrix manually when we change the input size. More precisely, whenever we increase the input size, we will initialize a new fully-connected layer and copy the previous network's weights to half of the current network. A visualization for 5×10 to 10×10 is shown in Figure 12, where we initialize the bottom half (shown in blue) of the weight matrix with the previous weights and randomly initialize the upper half of the weight matrix (shown in green).

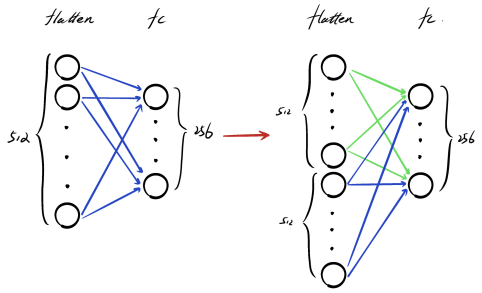


Figure 12: Grouped action example

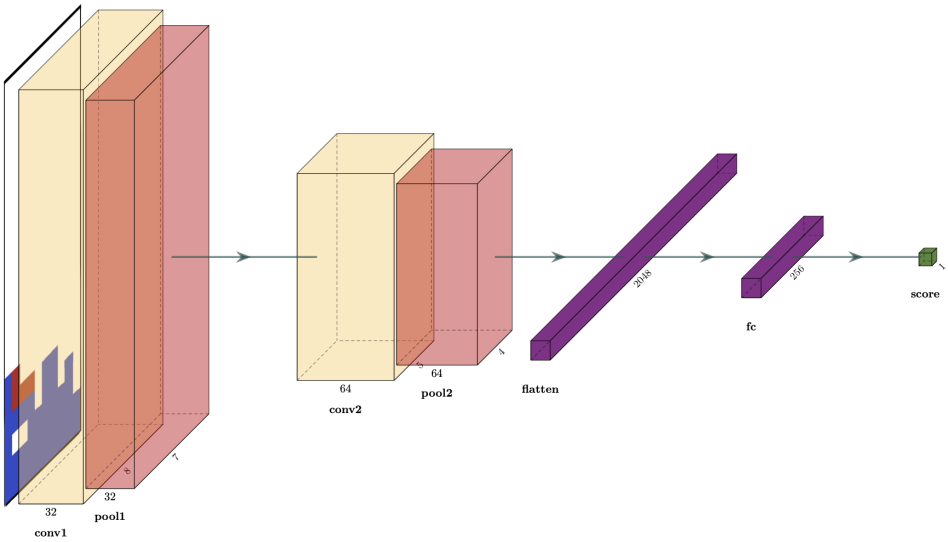


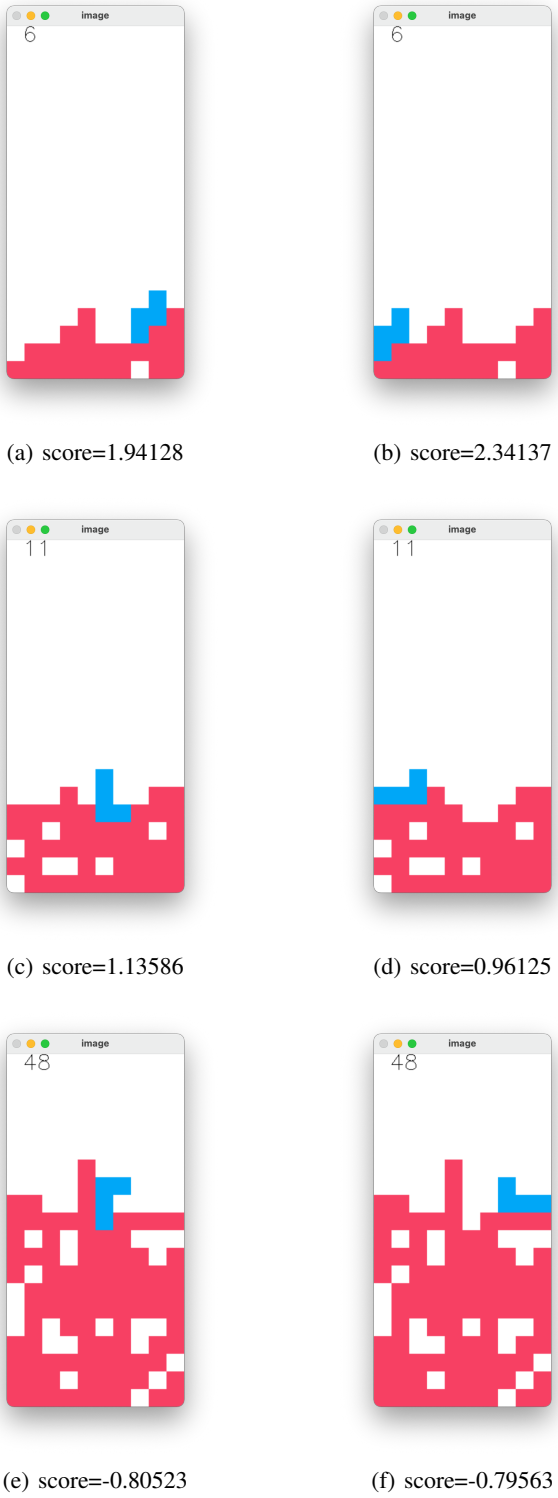
Figure 11: Deep Q-network architecture

Appendix C

features	min steps	avg steps	max steps
baseline	83	274.39	863
without BN	108	319.15	988
without DO	82	203.91	513
$\alpha = 1 \times 10^{-6}$	63	124.26	325
$\alpha = 5 \times 10^{-6}$	45	158.41	425
$\gamma = 0.85$	78	300.02	890
$\gamma = 0.95$	92	318.87	938
$\epsilon = 0.9$	50	232.81	734
$\epsilon = 1.0$	112	310.59	874
$\delta = 4 \times 10^{-4}$	56	258.59	437
$\delta = 6 \times 10^{-4}$	38	180.19	295
$BS = 128$	77	224.45	695
$BS = 256$	93	267.91	650
$BS = 512$	88	325.05	785
$BS = 1024$	59	311.92	798
$MS = 2000$	53	196.30	515
$MS = 5000$	55	212.99	612
$MS = 10000$	75	257.16	823
$MS = 20000$	103	312.34	919

Table 6: Tabular Results for RL Agent

Appendix D



(a) score=1.94128 (b) score=2.34137

(c) score=1.13586 (d) score=0.96125

(e) score=-0.80523 (f) score=-0.79563

Figure 13: Scores for some sample states