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

Abstract

Complete this section for D4.
The *Abstract* should be at most 150 words long, and should summarize briefly what your project is about. This includes the motivation for the problem (2-3 sentences), the problem you tackled (2-3 sentences), and your main results (1-2 sentences).

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**

  In brief, the methodology of this project consists of four main parts, 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.

  – 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. 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:

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

Table 1: Reward Function

- **Complete this bullet point for D4.**

  Emphasize your contributions. How should we interpret the results? Why should people care about this work? Does this project introduce any novel techniques or reveal any unexpected findings? In bullet point forms, list 3-4 key contributions of your project.

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 Gärtner, 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 trail-and-error approach. Every time when generating successors(states), we define a heuristic value associated with each state which is calculated based on a combination of weighted features. We would always choose successors with the lowest heuristic value at every step. For a handcrafted agent approach, we define 4 features:

1. number of lines cleared
2. number of holes
3. sum of difference of height between adjacent columns
4. max height

The goal is to figure out how much weight we should give to each feature such that the agent would have optimal performance. We define the performance as the most number of rows cleared before the game is terminated.

In a handcrafted agent approach, we want to intuitively give high weights to features which are more likely to cause game to be over and give low weights to features that are more likely to cause rows to be cleared. For example, we want to avoid choosing successors(states) which have a high max height. This is because intuitively, the higher the board is, the closer the board is to the "ceiling" (vertical limit of the board), which means there may be fewer steps to play before the game is terminated. As a result, we want to assign a high weight to this feature, which is max height. Similarly, we would also want to avoid choosing successors which have a high sum of difference of height between adjacent rows. This is also intuitive: if there are a lot of "gaps" in the board, it makes it harder to clear rows simply because the "gaps" are not easy to be filled perfectly. However, if a successor would result in one or more lines cleared, then intuitively we know that this is a good move and we want to assign low weight to this feature.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 value and use it in the next simulation until certain stopping criteria has been met.

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 "blindly", we are more likely to arrive at an optimal combination taking fewer steps.

Local Search Agent

Essentially, in a handcrafted agent approach, human decisions are used to adjust weight combination during every iteration of simulation. Such approach obviously has some limitations simply because a lot of possible adjustment to the weights can not be tested/explored. Moreover, the features that are used in a handcrafted agent approach, which has a number of 4, are also determined by human intuition. What if there are some more underlying features yet to be taken into consideration? Is there a more systematical way to choose successor and improve weight combinations other than just by intuition? With these questions bearing in mind, we propose a second approach which utilized local search and in particular, a genetic algorithm approach.

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

1. number of lines cleared
2. number of holes
3. sum of difference of height between adjacent columns
4. max height
5. number of blocks currently on the board
6. number of rows with at least a hole in it
7. max height difference between lowest column and highest column
8. number of cells that is not filled with no blocks above it and two horizontal neighbours 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 not no longer a hole

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

- Chromosome: An array of 10 numbers representing our weights combination, each number is ranged from 0 to 1
- Fitness: How many lines are cleared before the game is terminated
- Selection: After ranking each state based on fitness score, we would remove a portion of states with low fitness scores and arbitrarily combine any two states from the remaining states as parents for the next generation
- Crossover: Take a combination of weights from the first parent and the second and make it the chromosome for the children
- Mutation: After crossover, one random weight value in the chromosome of the child state is mutated with certain probability

The game would start by generating random chromosome for each possible child (the state after dropping the second block). For each of these child, we would run a new simulation on it until the simulated game terminates. We would then record how many lines have been removed in this game and use that as our fitness score for this child. Next we would begin our selection process which would give us a number of parent states. For each group of parents, we would perform crossover and mutation operations to get us children of 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 after certain number of generations have been created. We would then choose the child with the highest fitness score and use it chromosome as our final weight combination.

It is obvious to see that in this genetic algorithm approach, we would eliminate a lot of human assumptions. That is, what weight value should be changed and by how much. We would also take into consideration of more features which may further eliminate human assumptions of which feature could have an underlying impact on the game-play or not. Despite of 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 nine 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:

	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 2: 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.[1]

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

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.[2] We will stop Q-learning until the agent completes the game, and we will 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[3], which is particularly bad for training.
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.

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. [4] For example, if we got an L-shape Tetromino as shown in Figure 1, one potential *grouped action* will be $\{left \times 4,\ clockwise\ rotation \times 2,\ down \times 12\}$, which

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

[3]Notice that according to the Table 1, we will only get a reward when we place a Tetromino so we won't get anything during the placement process.

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

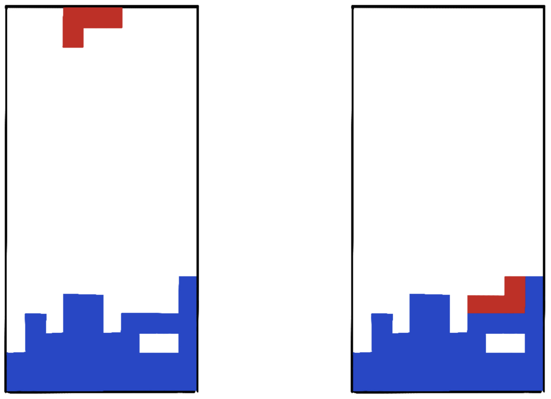

Figure 1: Grouped action example

is essentially the successor function from 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 into one-dimensional:

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

Table 3: 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 2.

Notice that 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. Here is a summary for each layer in Figure 2:

- **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.

On top of that, here is the pseudo-code to train our DQN:

In summary, DQN architecture successfully solves the problems of sparse reward and enormous states, which gives us an efficient and effective estimation for the original Q-learning in Table 2. 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 genetic algorithm.

Results

Environmental Setup

Evaluation Metrics

Missing first half of the evaluation metrics

One thing worth mentioning about the evaluation metric is that we will 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. 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.

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

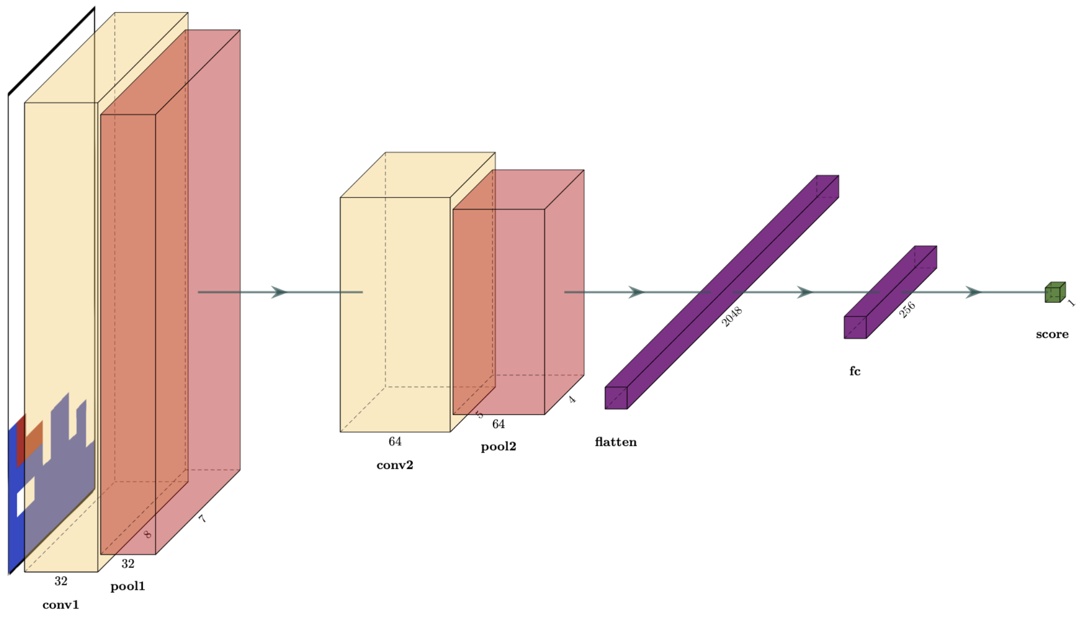

Figure 2: Deep Q-network architecture

```
1 Function train(cur, DQN):
2     Input: $cur$ is current game state and $DQN$ is the network, which supports two operations $predict$ and $fit$
3     Output: $nex$ is the next game state
4     $states \leftarrow successor(cur)$
5     $maxQ \leftarrow 0, reward \leftarrow 0, maxState \leftarrow$ None
6     for $state$ in $states$ do
7         $score \leftarrow DQN.predict(state)$
8         if $score > maxQ$ then
9             $maxQ \leftarrow score$
10            $reward \leftarrow reward(state)$
11            $maxState \leftarrow state$
12        end
13    end
14    $Q' \leftarrow (1-\alpha) \cdot maxQ + \alpha \cdot (reward + \gamma \cdot maxQ)$
15    $DQN.fit(maxState, Q')$
16    return maxState
```

Figure 3: Pseudo-code for Training

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

Table 4: Reward Function

(compared with cleaning a line) for safely landing a Tetromino for the sake of more frequent simulation for backpropagation. Meanwhile, this metric also introduces a substantial penalty for losing the game to produce strong negative feedbacks for the network.

Experimental Designs

- **Handcrafted Agent**

  The experimental design for the handcrafted agent is rather straight-forward.

  Every time when we start a new iteration of simulation, we would assign an adjusted weight combination to the game. Initially the weight combination is one that is resulted from an intuitive thought process. After the first simulation, we would try to adjust the weight of one or more features and supplement the new weight combination to a new simulation of game. For example, we would increase the weight of feature A (sum of difference of height between adjacent rows) while keeping other features the same to see whether or not this would yield a better result. We would repeat this process till certain threshold is reached. Possible thresholds are: number of simulations performed so far, trivial improvement (less than 1%), if not no improvement, has been witnessed for a consecutive of 10 simulations.

- **Local Search Agent**

  The experiment would mainly consists of 4 steps:

  1. Determine the fitness score of one state after running simulation with respect to one chromosome
  2. Delete portions of state with low fitness score and randomly choose parents from the remaining

3. Crossover process, and in particular, how to allocate portions of weights taken from each parent
4. Mutation process

There are some important details in step 1, 2, 3 and 4.

Firstly, in step 1, since each simulation is independent, the sequence of blocks generated in the game is not exactly the same. As a result, even though the chromosome of one state is fixed, we might arrive at different fitness scores in different simulations. To tackle this, instead of running one simulation for each state, we would run a total of 10 simulations. We would then record fitness scores for all 10 simulations and use the mean of them as the final fitness score for a given state and its associated chromosome.

Secondly, in step 2, after sorting each state based on their fitness score, we would first remove 1% of those with lowest scores. For the remaining states, which say, have a total number of N, then we would group two states randomly to form parents of next generation. There are in total $\binom{n}{2}$ ways of grouping. As for implementation, for each state we would define a set of visited other states called S_i where i is the index of current state. During the selection process, we would select state i to pair with state j if and only if $i \notin S_j$ and $j \notin S_i$

Thirdly, in step 3, when determining the exact combination of chromosome from the parent states, we would take into consideration of fitness scores of parents as well. It make sense: if one parent has higher fitness score than the other, then 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 say the fitness score of first parent is $f1$ and that of second parent is $f2$. Assuming that $f1 >= f2$, then we define the combination as follows:

– if $\frac{f1}{f2} = 1$, then we take first 5 weights from the first parent and last 5 weights from the second parent
– if $0 < \frac{f1}{f2} - 1 <= 0.25$, then we take first 6 weights from the first parent and last 4 weights from the second parent
– if $0.25 < \frac{f1}{f2} - 1 <= 0.5$, then we take first 7 weights from the first parent and last 3 weights from the second parent
– if $0.5 < \frac{f1}{f2} - 1 <= 0.75$, then we take first 8 weights from the first parent and last 2 weights from the second parent
– if $0.75 < \frac{f1}{f2} - 1$, then we take first 9 weights from the first parent and last 1 weight from the second parent

After this step, we have completed the selection and crossover step.

Lastly, in step 4, we would mutate one weight value out of ten with a probability of 10%. In particular, we would have a random number generator which only returns 0 and 1, with the probability of 90% and 10% respectively. If it returns 1, we would then have a second random number generator which generates an integer i from 0 to 9, each with a probability of 10%. We would then mutate the i^{th} weight value to be an arbitrary number from 0 to 1. After this step we have already created a new generation of children with new chromosomes.

- **Reinforcement Learning Agent**

  As mentioned above, since we implemented a Tetris environment, we can say that we have unlimited training data. Therefore, for each hyperparameter combination, we will train the network for 1,000 games, validate and test separately for 100 games, where we will measure both metrics during training but only calculate the primary metric for validation and test. As for the hyperparameter, we will leverage results from Stevens's paper (Stevens 2016) and initialize our hyperparameters using their suggestions as follow.

  – **Optimizer**: $RMSProp$
  – **Learning rate**: $\alpha = 2 \times 10^{-6}$
  – **Discount rate**: $\gamma = 0.9$
  – **Regularization**: Batch normalization and dropout with a retention probability of 0.75

  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.

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

  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. Note that according to the setups mentioned in the Methodology section, 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[5], 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 correspondingly as follow.
     Note that we need a describes ϵ because we want to explore more in the beginning (in our case, we will use $\epsilon = 0.5$), and as we approach the end of the training phase, we want to exploit more (which means $\delta = \frac{\epsilon}{\text{\# of training}} = \frac{0.5}{1000} = 0.0005$).[6]
  2. **Training by Part**
     According to the experiment results from Steven's paper, 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

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

[6]This gives two extra hyperparameters that need to be adjusted.

```
Function train(cur, DQN):
    $states \leftarrow successor(cur)$
    $maxQ \leftarrow 0, reward \leftarrow 0, maxState \leftarrow$ 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

    // $\epsilon$-greedy algorithm
    $p \leftarrow uniform(0, 1)$
    if $p > \epsilon$ then
        $maxState \leftarrow random(states)$
        $maxQ \leftarrow DQN.predict(maxState)$
        $reward \leftarrow reward(maxState)$
    end
    $\epsilon \leftarrow \epsilon - \delta$                // $\epsilon$ decaying

    $Q' \leftarrow (1 - \alpha) \cdot maxQ + \alpha \cdot (reward + \gamma \cdot maxQ)$
    $DQN.fit(maxState, Q')$
    return maxState
```

Figure 4: Pseudo-code for Training with ϵ-greedy

will solve this problem by training our network part by part, where 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. However, the sequential training process is 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 5, 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).

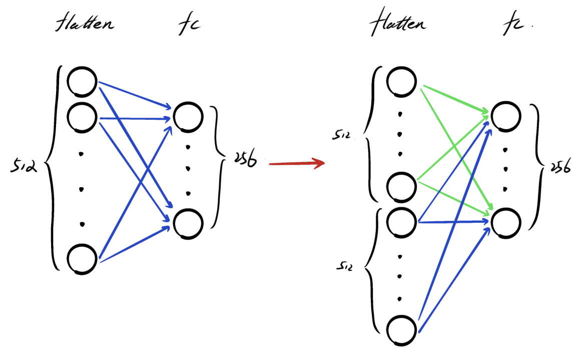

Figure 5: Grouped action example

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.

Complete the following two paragraphs for D3.

Describe the findings from your evaluation. Describe both (a) how well your techniques worked, and (b) what you learned about the problem through these techniques.

Prepare figures (e.g., Figure 6) and tables (e.g., Table 6) to describe your results clearly. Make sure to label your figures and tables and explain them in the text. If you are comparing the performance of algorithms, include statistical tests to assess whether the differences are statistically significant. If possible, describe how your techniques compare to prior work.

Techniques	F-1 Score
Baseline	0.80
Another Baseline	0.76
My Awesome Algorithm	**0.95**

Table 6: example of a table summarizing the results

Discussion

Complete this section for D4.

The *Discussion* section ($\sim$1 pages) describes (a) the implications of your results, and (b) the impact and the limitations of your approach.

For the results, describe how a reader should interpret them. Try to form concise take-away messages for the reader. For your approach, describe the extent to which your

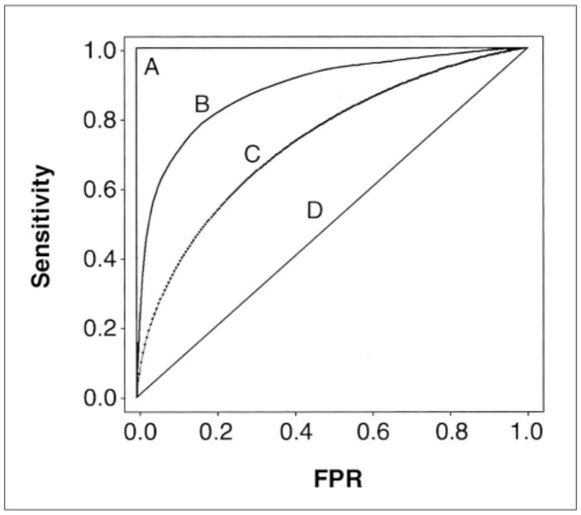

Figure 6: ROC curve of my awesome algorithms

approach helps to solve the problem. Describe any limitations of your approach. If possible, compare your results and your approach to that of prior work.

Conclusion

Complete this section for D4.

The *Conclusion* section ($\sim$0.5 pages) provides a brief summary of the entire paper. In this section, describe

- the motivation, the problem, and your results, and
- 3-4 promising future directions.

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):194–200.

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.