NTNU
Kunnskap for en bedre verden

Department of Computer Science

IDATT2502
Applied Machine Learning

Feasibility of Tetris w/ DQN

Author:
Erling Sung Sletta

November, 2021

Contents

1 Introduction

This report is based on a project conducted as the final examination and curriculum in the subject IDATT2502 - Applied Machine Learning with Project. Of several proposed project ideas, the one chosen for this project based itself on using Reinforcement Learning to solve a given environment. More specifically, this report will be going over the process, results and conclusion of my attempts at implementing an algorithm to train an agent that can play Tetris using Deep Q-Learning. Additionally, we will look at an alternative method using a genetic algorithm for comparison sake.

1.1 Personal Motivation

For our eighth obligatory exercise we used Reinforcement Learning to solve the so-called CartPole problem, in which it the goal is to prevent a pendulum attached to a cart from falling over. The concept of teaching a machine to learn and adapt to it's environment to figure out what it should do left an impression on me, so when I was then presented with the idea of attempting to solve more complex environments, like Super Mario or Tetris, I was quickly tempted. While the other topics were all quite interesting as well, my video-gaming background simply wouldn't let me pass up this opportunity.

1.2 Tetris

Tetris is a video game created by Alexey Pajitnov 1984. While the original game might be old, this timeless classic has had numerous adaptations and remakes throughout the years, easily making it as one of the best-selling video games of all time[9]. The games concept is quite simple; the player is presented with a 10x20 grid, and receives one of seven differently shaped pieces called tetrominoes. The players goal is to place these tetrominoes in such a way that whole rows are filled, which in turn clears the line and rewards the player with points.

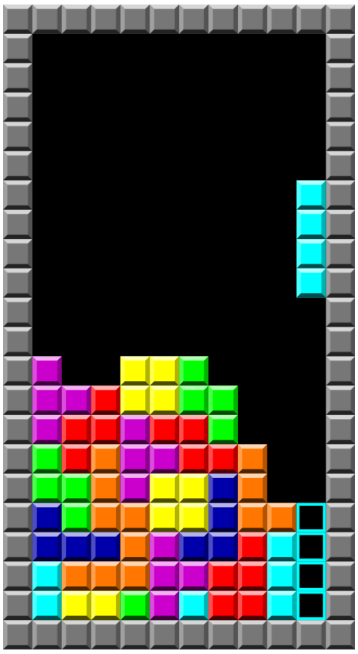


Figure 1: A typical Tetris game screen[9]

2 Background

Now that you've been introduced with the game itself and the end-goal of this project, we can start looking into how it can be achieved. Before getting into the specifics, the following sections will provide some context on the algorithms that were used for this project.

2.1 Deep Q-Learning

2.1.1 Q-Learning

Deep Q-learning (henceforth DQN) is a variation of the reinforcement learning algorithm known as Q-learning; an algorithm that doesn't use a model of the environment, but instead keeps a table, often called a Q-table, to keep track of each state + action combination, and a value that represents how favourable said action is for said state. The agent then uses this table as the basis for an algorithm to be used in the policy.

$$Q^{new}(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t,a_t)}_{\text{old value}}\Bigg)}^{\text{temporal difference}}$$

Figure 2: Q-learning algorithm[10]

The algorithm above is used to calculate how good an action a_t is given when in state s_t. The learning rate α decides how much should be learnt at each step, in other words how quickly the Q-value should change. The discount rate γ decides how 'greedy' the agent is, meaning how likely it is favour immediate gain over long-term gain.

2.1.2 DQN Differences

DQN replaces the Q-table from Q-learning with a convolutional neural network. This network takes the state and "analyses" it to find and learn patterns of which actions lead to which rewards, given said state. The networks output is some amount of values, each representing an action in the environments action space. The action represented by the best of these values, typically the largest, is then the action the agent chooses to take.

To keep track of the reward gained after performing a given action at a certain state, a technique called ***experience replay*** is used. This servers as the agents "memory", and by using it, the agent can avoid being biased by finding correlations between recent sequential actions and observations. Instead, the memory is randomly sampled at regular intervals, and these samples are used to update the Q-values in a similar fashion as with normal Q-learning.

2.2 Genetic Algorithm

Another algorithm that was used in this project is a genetic algorithm. This algorithm bases itself on the real-world phenomenon of natural selection. The general idea is that we will iterate over several generations of Tetris agents, and each time; only a select number of the strongest players will survive. The strongest player could simply enough be the one that cleared the most lines, but if that's our only trait then what is there to pass on the the next generations? This next section will cover how we can give our agents individual traits so that they will perform differently.

2.2.1 Tetris Heuristics

Heuristic techniques are used to quickly and efficiently find a solution to a problem, one that doesn't necessarily have to be the best possible, but sufficient nonetheless. There are many published research papers on the usage of heuristics in Tetris. Max Bergmark wrote a thesis in which they discuss board sustainability and breadth-first search heuristics[1]. Bergmark writes that to evaluate the sustainability of a game board, an agent could for example calculate the number of holes on said board for each possible placement. Using some scoring system, the agent can then have a more concrete generalization of the properties a sustainable board should have, without having to consider the exact state of the game and all possible future outcomes.

2.2.2 Evolution

To further improve the agent's scoring system, Bergmark defines three different categorizations of holes, and adds some weighting function to each of them. The final score is then given by the sum of each weighted hole. These weights are somewhat arbitrary, as they are constant and defined by Bergmark directly, but this is where we can make use of machine learning.

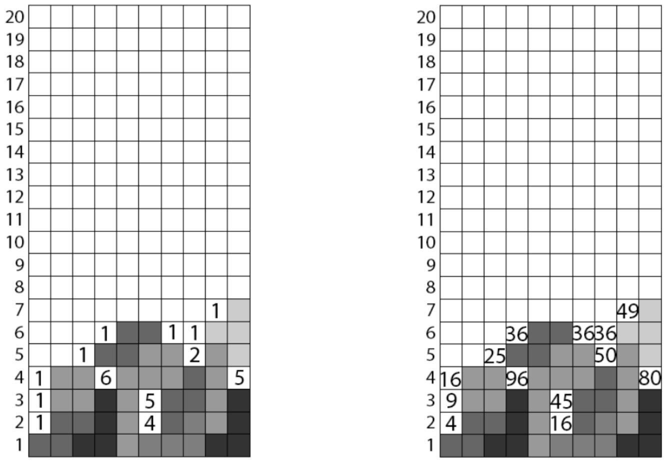


Figure 3: Bergmark's examples of hole weighting[1]

We can define an agent as a member of a population, and the weights as the agents individual genes. In a population of some number of agents, each agent has a different set of genes, and are then left to play the game for a some defined amount of time, and when they're finished we can say that a generation has passed. The agents who achieved the highest score during their generation are sorted out as the "strongest" and are moved on to the next generation. To fill out the remainder of the population, the survivors are crossed over, meaning that their genes are randomly passed on to new agents. To add diversity, a certain chance for mutation is set, meaning that a new agent is given a randomly assigned weight. This adds some diversity to the population and prevents it convergence to a non-optimal solution that could take place if for example the optimal gene wasn't present in the first population.

3 Related Work

3.1 OpenAI

OpenAI is an AI research laboratory whose ”mission is to ensure that artificial general intelligence benefits all of humanity”[6]. One of OpenAI’s goals is to standardize environments that are used in AI research, and one of their contributions towards this goal is the open source Python library known as Gym. This Reinforcement Learning toolkit aims to supply users with environments that are simple to create and interact with[3]. The following bit of code shows an example of how the previously mentioned Gym CartPole environment can be setup and used.

```python
import gym
env = gym.make('CartPole-v1')

# env is created, now we can use it:
for episode in range(10):
    obs = env.reset()
    for step in range(50):
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)
```

Some particularly noteworthy variables here are ’observation’, ’action’ and ’reward’. The ***observation*** is often called state, as it represents the current state of the environment. In the CartPole environment these are values representing the position and velocity of the cart and pendulum. An ***action*** is one of some amount of possible actions that the agent, in other words the player, can take. In this environment it is whether the cart should move left or right. Here the action is decided randomly, but in actual implementation, policies are defined to select an action based on some algorithm. After an action has been made, the agent gets a ***reward*** based on the result of said action.

3.2 Reinforcement Learning

Using Reinforcement Learning to play video games is a much researched topic. Even the specific, albeit simple, Google search ”Tetris DQN” will net you a multitude of theses, GitHub repositories and so on. As part of a publication series explaining deep reinforcement learning[8], Jordi Torres describes the process of creating an algorithm to train an agent to play the classic Atari arcade game Pong[7]. Torres uses an OpenAI Pong environment that emulates an Atari 2600 environment, offering a resolution of 210x160 pixels with 128 colors. An agent does not get access to the games underlying state, but has to work with the rather complex raw pixel data. To circumvent this, Torres applies a series of processing-steps to simplify this data into smaller more practical states for us to work with, a method that greatly helped the progress of this project.

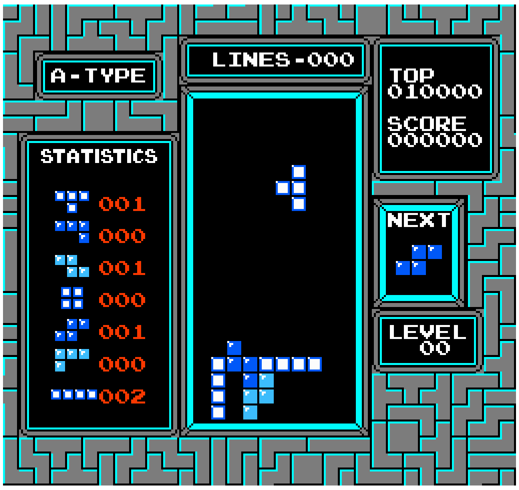
Torres also mentions a 2015 project led by DeepMind, in which a group of researchers attempt to ”create a single algorithm that would be able to develop a wide range of competencies on a varied range of challenging tasks”[5]. The group successfully developed an architecture that performs above human level in most of the tested environments, and even several times better than a professional in a large portion of them.

4 Method

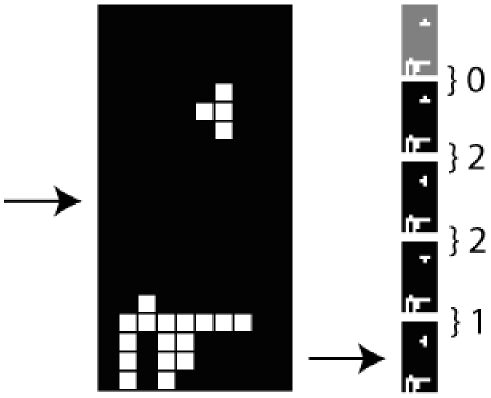
4.1 Environment

4.1.1 Specifics

The environment used for this project is an OpenAI Gym interface that emulates classic Tetris on the NES, developed by Christian Kauten[4]. This particular environment is similar to the previously referenced Pong environment, in the sense that it does not let our agent directly view the games underlying state, with a few exceptions. The state returned by the environment has the shape 240 x 256 x 3, representing a 240x256 pixel image with RGB color. This is a rather complex structure that would be very taxing to process, which is why a method to reduce the resolution drastically is used. The method is heavily inspired by Torres' Pong solution[7], and reduces the state to some number of 10x20 images, each representing the Tetris board at sequential time steps.



(a) The original state



(b) Processing steps (image upscaled)

4.1.2 Processing

In figure a we see the visual representation of the state that the environment initially returns. This initial state would be very inefficient to work with, as it is not only very complex; but also has a lot of redundant noise. For example, the agent does not need to see the tetrominoes scattered around in the background, or the various texts around the board. Not only do these details increase the state complexity, but they could also interfere with our learning algorithm.

For these reasons, the image is processed to simplify the structure and crop out the noise, as we see in the first step of figure b. The image is then resized into a 10x20 image, represented by an array of ones and zeroes. Since the agent would have no dynamic piece knowledge with just a single layer, we keep each processed state in a buffer so we can pass this on to the agent. In the figure we see the newest image being appended to a buffer with a capacity of four states, as well as the actions that were taken between each state. Below is a table showing which action each number represents.

Action Space	
#	Description
0	Do nothing
1	Rotate piece clockwise
2	Rotate piece counter-clockwise
3	Move piece right
4	Move piece left
5	Move piece down

4.2 DQN Implementation

4.2.1 First iteration

The first DQN-model used in the project project did not use the board state, but instead supplied the agent different combinations of the following variables: their currently held tetromino, it's rotation, the number of lines it had cleared, current score, next tetromino and the current height of the board. For rewards, the agent got rewarded for clearing lines and/or staying alive, and penalized when the game ended, and for some runs when the board height increased.

Exploration rate for this model was dynamic, given by $\max(\varepsilon_{min}, 1/(\varepsilon_{decay}^{n}\ t/250))$, where n represents the current episode, and t the current time step. ε_{min} was set to 0.01, while different values of ε_{decay} were used, the final one being 2.5.

For optimization, the model used RMSprop with smooth L1-loss. These weren't chosen for any particular reason, other than being familiar. This models iterative process didn't last long enough to consider about trying other alternatives, as it was quickly discarded in favour of the one described in the next section.

4.2.2 Second iteration

After realising what the 240x256x3 state represented, and finding a way to reduce it's complexity, a new DQN model was quickly developed to accommodate the new state. In the **Environment** section above, the image processing method was introduced and it was revealed that the final model uses states of shape 10x20xN, with N representing the number of state frames being buffered. Initially, however, the model used 20x20 frames. It was later revealed that this could cause result in lossy input and otherwise undesirable behaviours, so the image stretching was discontinued.

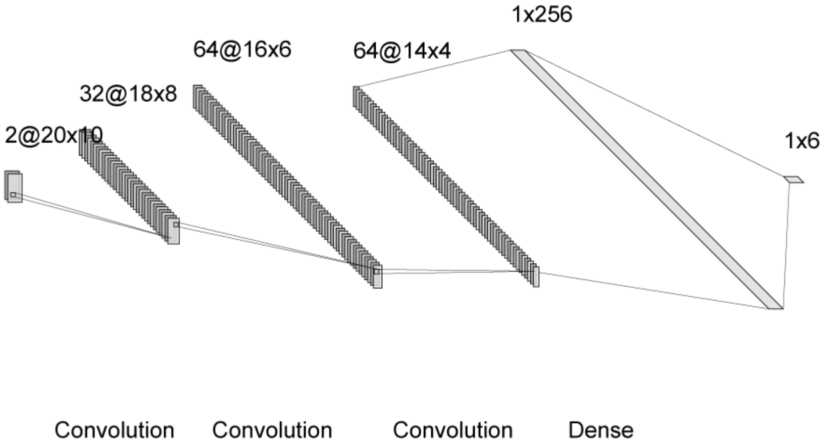


Figure 5: Illustration of the neural network used

Additionally, this model keeps a copy of it's neural network for usage when updating parameters to counteract instability caused by trying to estimate errors using subsequent near-equal steps. The copy network is kept stable, and only updated every so often on regular intervals.

For the reward scheme, similar experiments to the ones ran with the first iteration of DQN were used. Various attempts at rewarding for line clears and/or staying alive, and penalizing for build height, game over, etc. For optimization, Adam was used with MSELoss.

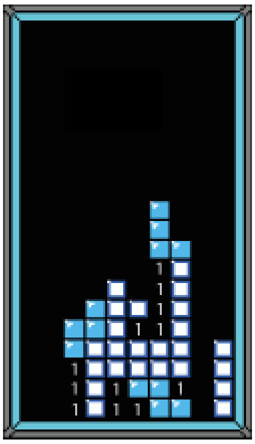
Hyperparameters used in this model at the final stage were $\alpha = 10^{-4}$, $\gamma = 0.9$, $\varepsilon_{start} = 1.0$, $\varepsilon_{decay} = 0.999985$, $\varepsilon_{min} = 0.05$, but different values were tried throughout the project. The same goes for the capacity and batch size used with the experience replay. Explorational rate was defined by the largest of $\varepsilon_{start} \cdot \varepsilon_{decay}^{t}$ and ε_{min}.

4.3 Genetic Algorithm Implementation

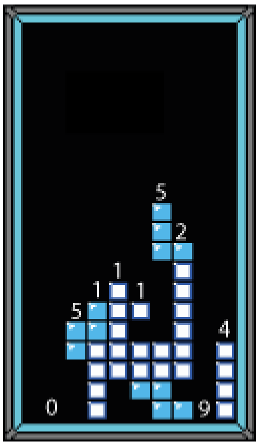
4.3.1 Heuristics

The projects source code implements methods to analyze the number of holes, lines cleared, bumpiness, aggregate height and largest height difference of a given board state. These traits are used to calculate the states sustainability after every possible move that can be made with a tetromino.

Holes are all weighted equally, and defined as any open slot that has a block any number of tiles above it. Lines cleared are simply any row that has been filled with tiles, and bumpiness is the sum of height differences between each adjacent column in the board. Aggregate high is the sum of the highest piece in each column, while largest height difference is the difference between the highest piece in any column and the lowest piece in any column.



(a) Holes



(b) Bumpiness

Despite having implementations, most of these analytical methods were not used while training for simplicity's sake. When scoring sustainability, the equation used was $w_1 \cdot \textit{lines\_cleared} - w_2 \cdot \textit{holes} - w_3 \cdot \textit{bumpiness}$, with w_1, w_2, w_3 each representing one of a given agent's weights. The state example above would get a score of $w_1 \cdot 0 - w_2 \cdot 12 - w_3 \cdot 27$.

4.3.2 Algorithm

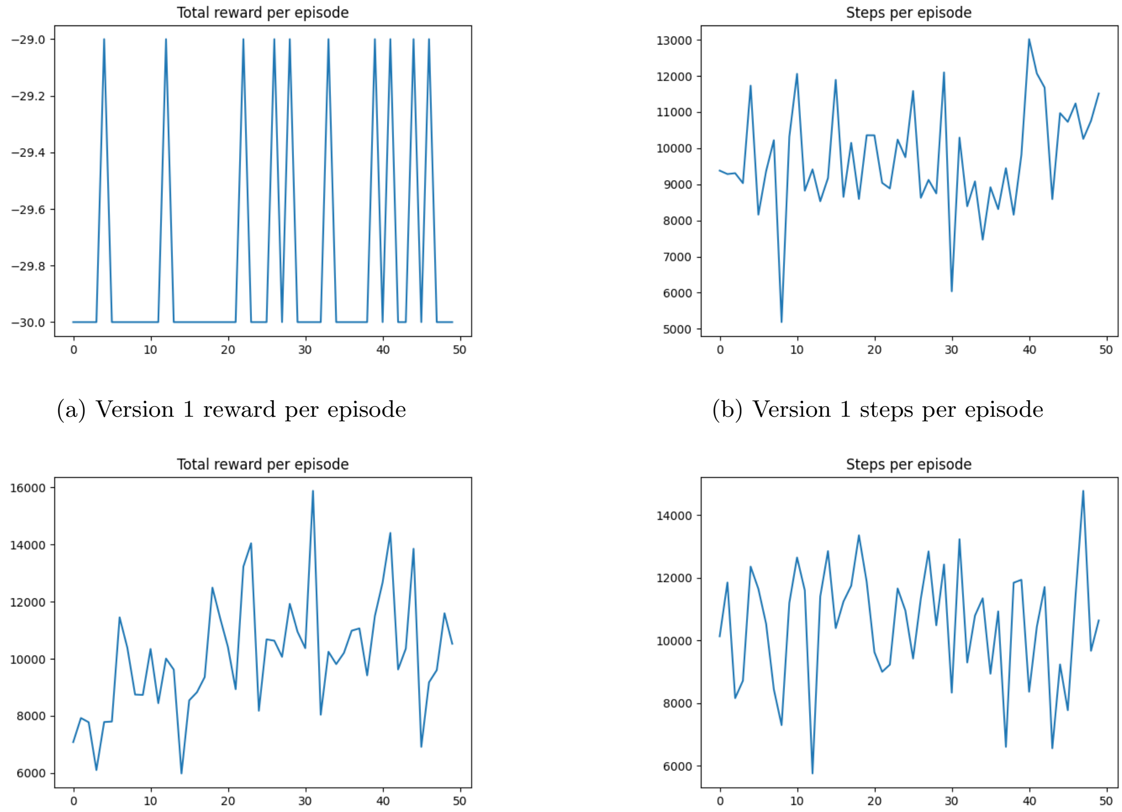
The implemented genetic algorithm does not differ much from the described earlier in the report. Some value is given to set the population size, number of generations, chance of mutation, max games and moves played, and the population percentage to change out per generation.

An example could be that a population of 8 is trained over 5 generations. To begin, 8 agents are created with randomized weights. They then play 5 games of Tetris, with a maximum of 50 moves per game. Once all agents have finished their games, the players are sorted based on how many lines they managed to clear over the course of their games. This marks the end of the first generation. The best scoring 30% of the players are then kept and transferred over to the next generation. To fill the remaining 70%, random samples are made of the kept players from the previous generation, and they're crossed. This means that a new agent is made, inheriting the weights of it's parents. Each weight has a 50% chance to come from either parent, and there's also a 5% chance that a weight could mutate, meaning that it is randomised instead. This process is then repeated, until all generations have passed and we're left with our best possible population.

5 Result

5.1 DQN Models

5.1.1 First iteration



(a) Version 1 reward per episode

(b) Version 1 steps per episode

(c) Version 2 reward per episode

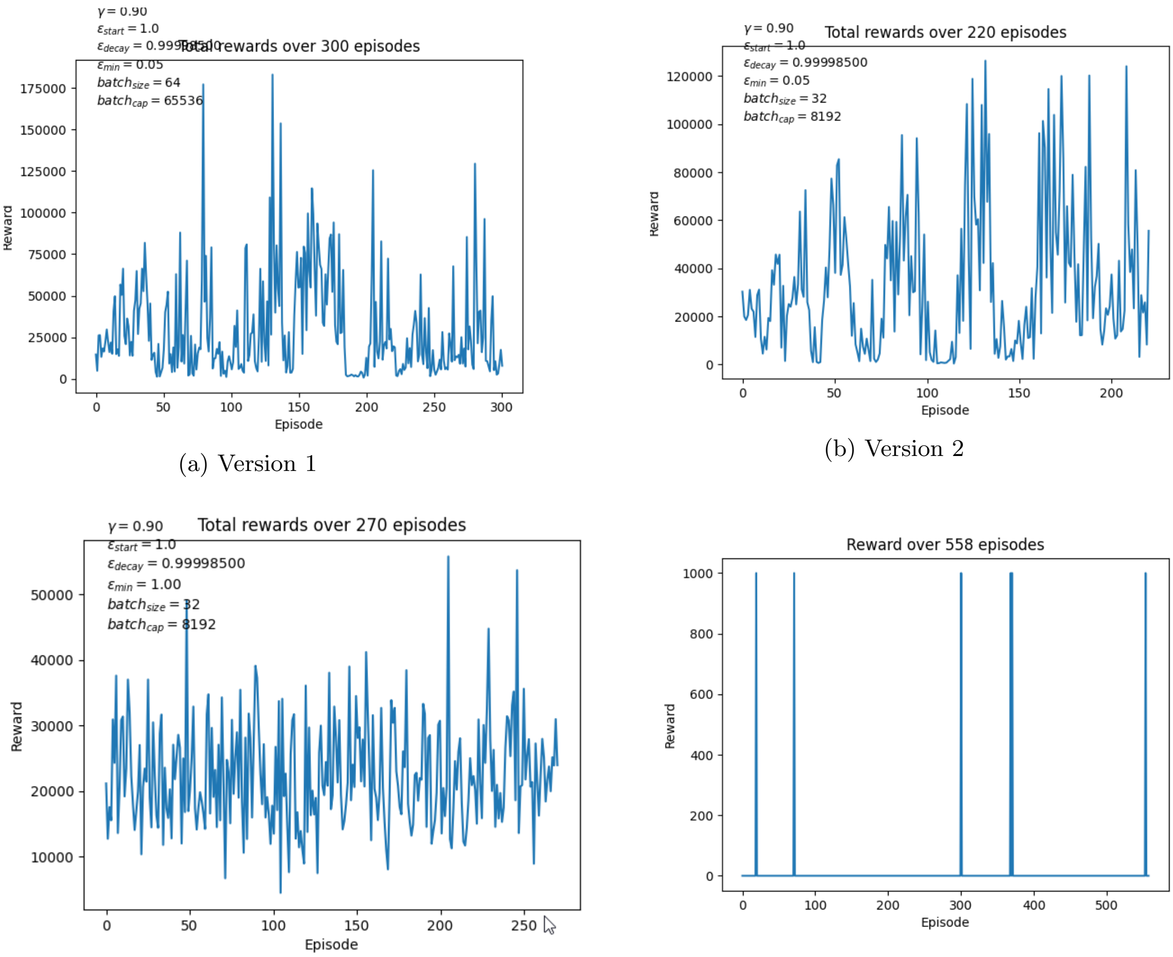
(d) Random from second iteration

Figure 7: Rewards with DQN using informational state

Version 1 simply gets rewarded for clearing lines, and penalized when the board height increases and when the game ends.

Version 2 gets rewarded for clearing lines as well, but this version also gets rewarded for staying alive longer. This version does not get penalized for building higher, but does lose points when the game ends.

5.1.2 Second iteration



(a) Version 1

(b) Version 2

(c) Random

(d) Version 3 reward per episode

Figure 8: Rewards with DQN using image state

Version 1 and 2 are practically identical. The only difference lies in their batch size and memory capacity. Version 1 has a batch size of 64, with a capacity of 65536. Meanwhile, version 2 has a batch size of 32, and a capacity of 8192.

Version 3 was rewarded 1000 points for every line clear, without any penalties.

Episodes using this line of models typically ended one of two ways after training for a while;

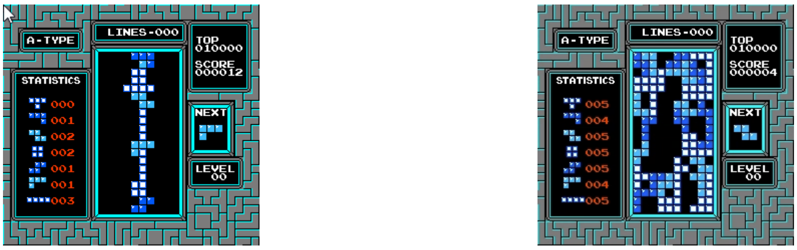


Figure 9: Endscreen from version 1 episode 60 and 80

5.2 Genetic Algorithm

5.2.1 Random Heuristics

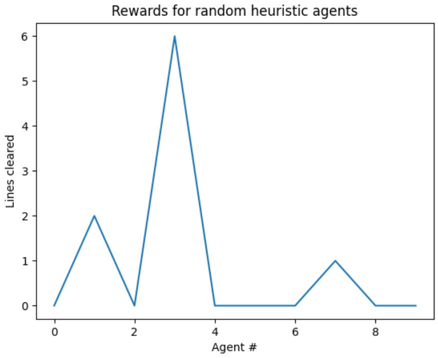
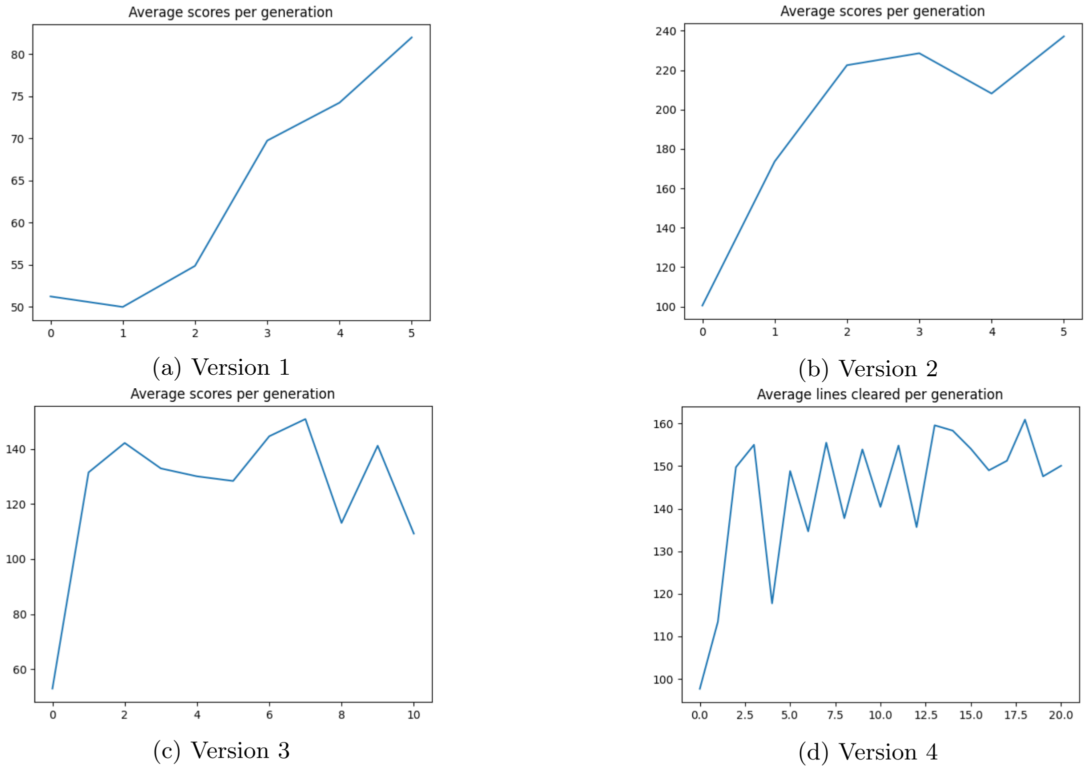


Figure 10: Rewards for 10 randomly initiated agents

5.2.2 Evolutionary Heuristics



(a) Version 1

(b) Version 2

(c) Version 3

(d) Version 4

Figure 11: Average line clears per generation with genetic algorithm

Version	Generations	Population	Games	Moves
1	5	8	5	50
2	5	12	5	150
3	10	16	3	250
4	20	12	10	50

All versions used a mutation chance of 5%, and the best 30% of each generation was transferred to the next generation.

5.3 Comparison

It's clear that the genetic algorithm using heuristics yielded significantly better results, and in much smaller time frames. While the DQN models struggled to clear even a single line after long training sessions, first generation agents with random heuristics were already clearing several in almost every game right from the beginning. This difference is only widened over time, as the agents get significantly better in just a few generations time.

6 Discussion

6.1 DQN

6.1.1 First iteration

Version 1 does not seem to learn anything worthwhile. There is no apparent pattern in the amount of steps it stays alive, and the change we see in figure 7a is caused by the environment being able to terminate at both 19 and 20 blocks, depending on the height of the tetromino that ends the game. Looking at figure 7c it might look like the agent learns *something*, as the episodes seem to last longer over time, but when comparing with 6d the difference is barely noticeable.

6.1.2 Second iteration

Figure 8d makes it abundantly clear that the model struggles to learn much when it's goal is to clear lines. On the other hand, when the model is given a clearer, more easily obtainable goal, like staying alive, it does seem to improve somewhat. We can see this in figures 8a and b. Comparing these with 8c shows us that the agent does indeed learn, and has several instances where it reaches noticeably higher rewards than that of the random agent. It can, however, be observed that the model is still quite unstable, as we see by the several dips in rewards over several episodes.

6.2 Genetic Algorithm

6.2.1 Random Heuristics

Figure 10 shows the lines cleared during one game for 10 agents initiated with random weights. As we can see, the performance of individual each agent is very different from another, as one could expect. It's clear that using heuristics does give us a decent framework, but it's arbitrary performance is not satisfactory. In addition, if one were to include more analytical features, the chance of getting good random weights is lowered.

6.2.2 Evolutionary Heuristics

The genetic algorithm shows extreme improvements in the overall performance of the heuristic agents. It can be observed across the board that the populations average drastically increases within the first few generations. The rewards do seem to plateau, but this can be explained by the limitations of having a set number of games with a set number of moves. Another explanation could be faults with the environment and code implementations, but these will be handled in the upcoming section **Sources of Error**.

In most of the sub-figures in figure 11, and especially 11d, rhythmic dips in the generational average can be observed. These could possibly be caused by bad mutations, as the numbers would add up. Version 4 has a mutation chance of 5%, which means that one mutation can be expected to occur once for every 20th agent. With a generational population of 12, it would make sense for a mutation to happen roughly every other generation. Furthermore, figure 10 shows that it's not particularly uncommon for an agent to get bad random weights.

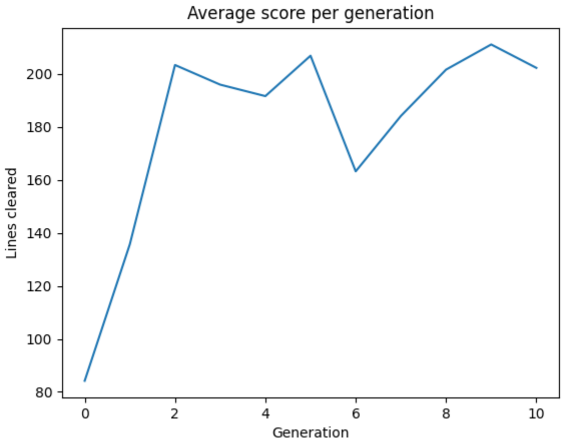
6.3 Comparison

Tetris is a game that follows Bushnell's Law, it's easy to learn and difficult to master[2]. While the controls and concept are simple the inherent structure makes the game very complex. Already on the first move there are roughly $7 \cdot 4 \cdot 8$ possible permutations, given 7 different pieces with 4 different rotations, which each can be placed in about 8 different positions. Bergmark calculated that the time complexity of a breadth first prediction algorithm has a complexity of $O(162^n)$, with n being the number of future tetromino whose possible permutations are calculated. A game using four step prediction could then take 28.1 years to complete. complete[1].

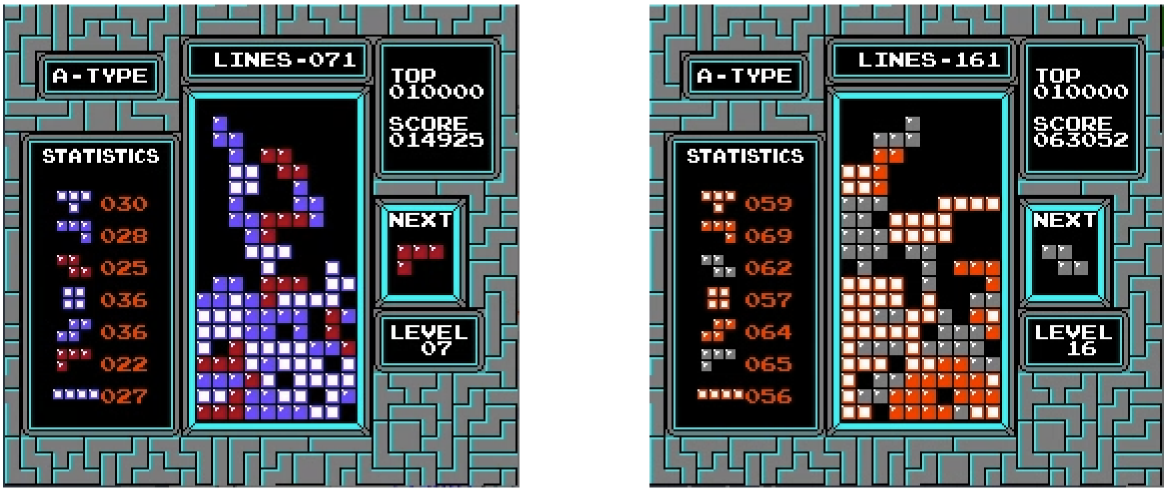
This complexity is what makes it so difficult for us to get a good result using DQN. There are simply too many possible states to consider. We've seen that the DQN models could learn when given specific tasks, like trying to stay alive, but trying to teach it to clear lines is difficult. This could be a direct consequence of the games complexity, in the sense that accidental line clears are few and far between. Therefore, the agent does not get much to work with. On the other hand, when we supply our algorithms with more general analytical tools that don't need to accommodate every single detail of our state, we see a great improvement in performance, as we've seen with our heuristic approach.

6.4 Sources of Error

During the writing of this report, several small yet consequential bugs were found that hindered certain methods from functioning as expected. One particularly devastating oversight was that since the environment changes the tetrominoes colours depending on the level, certain darker shades weren't being picked up and recognized by our agents. This didn't affect the DQN models too much, since this issue didn't happen before level 7, but the genetic algorithm definitely reached a score ceiling before it should have. The following graph is from a training session with the genetic algorithm. Parameters are the same as those in figure 11b, except with twice as many generations.



Unfortunately, 150 moves is not enough to get past level 7, so the performance change is not visible on the graph. It can, however, be noted that the environment increases the level every 10 line clears. This means that level 7 is reached after 70 line clears, so an optimal score before the issue appears would be a little over 70 lines. The agent trained on the new fixed patch, however, reaches up to level 16, meaning it's cleared 160 lines in a single game.



(a) Old

(b) Revised

Figure 12: Genetic algorithm finalist endscreens

Due to some shortcuts in code, tetromino placements are not as efficient as they can be, which leads to agents struggling to keep up with the falling speed at level 16.

7 Conclusion

As shown in the previous sections, performance is much better when the machine is supplied with concrete general definitions for what a good move looks like. Theoretically, it should be possible to have an agent know every single possible permutation state within Tetris, and thus every optimal move. Considering the complexity of Tetris, however, it is simply not feasible to create a pure DQN algorithm that can play Tetris effectively at a high level of play with current-day technology.

References

[1] Max Bergmark. ‘Tetris: A Heuristic Study : Using height-based weighing functions and breadth-first search heuristics for playing Tetris’. In: 2015.

[2] Ian Bogost. *Persuasive Games: Familiarity, Habituation, and Catchiness.* URL: https://www.gamasutra.com/view/feature/3977/persuasive_games_familiarity_.php (visited on 15th Nov. 2021).

[3] Greg Brockman et al. *OpenAI Gym.* 2016. eprint: arXiv:1606.01540.

[4] Christian Kauten. *Tetris (NES) for OpenAI Gym.* GitHub. 2019. URL: https://github.com/Kautenja/gym-tetris.

[5] Volodymyr Mnih et al. ‘Human-level control through deep reinforcement learning’. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[6] OpenAI. *OpenAI Charter.* URL: https://openai.com/charter/ (visited on 24th Nov. 2021).

[7] Jordi Torres. *Deep Q-Network (DQN)-I.* URL: https://towardsdatascience.com/deep-q-network-dqn-i-bce08bdf2af (visited on 16th Nov. 2021).

[8] Jordi Torres. *Deep Reinforcement Learning Explained.* URL: https://towardsdatascience.com/tagged/deep-r-l-explained (visited on 16th Nov. 2021).

[9] Wikipedia contributors. *List of best-selling video games — Wikipedia, The Free Encyclopedia.* [Online; accessed 26-November-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=List_of_best-selling_video_games&oldid=1056795783.

[10] Wikipedia contributors. *Q-learning — Wikipedia, The Free Encyclopedia.* [Online; accessed 25-November-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Q-learning&oldid=1056775822.