university of groningen / faculty of science and engineering

The effect of state representation in reinforcement learning applied to Tetris

Bachelor's Thesis

Gijs Hendriks, s2410540, gjghendriks@gmail.com
Supervisor: dr. M.A. Wiering

Abstract: Reinforcement learning (RL) is a paradigm within machine learning where agents try to maximize their reward. They do so by making decisions based on the representation of the current state of the world. The state representation is an important factor in the performance and training time when applying reinforcement learning to a problem. A representation can encapsulate different degrees of information about the world. In this paper the effects of different state representations and combinations of state representations are compared. This is done for the classical game Tetris using the standard temporal difference learning method. The experiment shows that representations with redundancy built in achieve the best results of ~ 23 lines cleared, while other representations with more information perform worse. In comparison to similar RL systems in literature this is a decent result, however other, non-RL methods, perform even better.

1 Introduction

Machine learning has almost become synonymous with artificial intelligence (AI) because of its abundance and success. This upward trend of success can be seen in the complexity of the games that have been tackled by machine learning over time. Examples include Backgammon by Tesauro (1994) and the famous Deep Blue machine for playing Chess by Campbell, Hoane, and Hsu (2002). More recent examples include systems for even more complex games like Go by Silver et al. (2016), Montezuma's revenge by Salimans and Chen (2018), Starcraft II by Vinyals et al. (2019) and Dota 2 by the openAI team (2019). These systems have achieved great results of expert-level players. Machine learning has also been applied to a simplified version of Tetris by Melax in 1998 [1]. While the goal of AI is not to solve games, they can serve as proofs that these techniques work in increasingly complex scenarios.

Reinforcement learning (RL) is a powerful paradigm within machine learning where agents try to learn a policy to maximize their reward. One of the reasons that makes RL interesting is because it does not need to be fed labeled data but instead learns from exploration or examples. This makes the paradigm extremely scalable in comparison to the more popular supervised learning where sometimes millions of hand-labeled data points are needed. Therefore the supervised learning approach is not viable if a general AI is to be found.

One thing that holds RL back is its need for extreme training times for complex tasks. From the openAI team (2019): "After ten months of training using 770 ± 50 PFlops/s·days of compute ...". Researchers are therefore on an eternal quest to decrease training time and increase performance. This is why it is vital to understand the fundamentals of RL and their impact on performance.

1.1 Tetris

Tetris is played on a board of cells arranged in an array of 10 by 20. Each cell can either be filled or empty. A Tetris block is called a tetromino (see figure 1.1) and can be described by a letter. A tetromino is selected randomly from a uniform distribution and is spawned at the centre top of the board. The player or agent can move the tetromino left or right and rotate the tetromino clock-

[1] http://melax.github.io/tetris/tetris.html

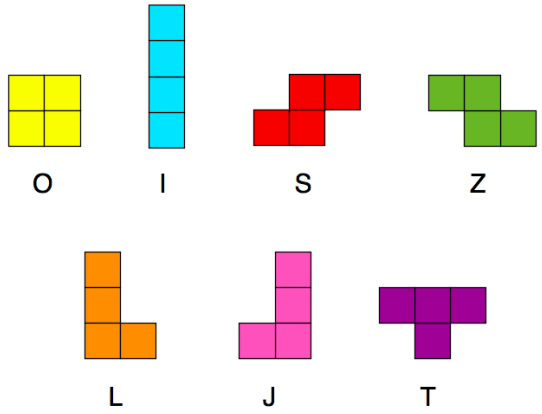

Figure 1.1: The different available tetrominoes and their corresponding letter.

wise 90 degrees. At a set time interval the tetromino moves down a row until it reaches the bottom of the board or another tetromino. When this happens, the tetromino becomes unmovable and a new tetromino is spawned at the top of the board. The cycle is then repeated. The goal of the game is to stack and rotate tetrominoes in such a way that rows become completely filled. When a row is completely filled, the row disappears and all the rows above, fall down. For each filled row, a point is scored. The game ends when incomplete rows stack up until the ceiling is reached.

1.2 State representation

One of the factors that contribute to the performance of an RL system is the state representation. A state representation is a model of a system at some time, for example: a state representation of Chess could be where all the pieces are and who's turn it is. The representation of the state can fully encapsulate the state of the game, but it can also be more abstract or incomplete, or even have redundant information. The state representation is important because it highly influences the system's design and training time. It is always a trade-off between redundancy and specificity. The naive approach where the representation is the whole environment (i.e. all the pixels on the screen) is slow to converge as shown by Andre and Russell (2002) and Dietterich (2000) and abstraction can improve training time without impacting performance negatively.

To research state representations, Tetris was chosen as a domain. Tetris is one of the best-known and most sold games of all time according to its creator[2]. It has a large state space: almost $2^{200} \approx 10^{60}$, because it has a grid of 10 x 20 and each cell can be two states: on and off. However, each row can not be completely filled so this makes the space a little bit smaller. To make matters worse, there are a lot of options to go from one state to the next. This is called a branching factor. For Tetris this branching factor is 9 - 39 depending on the state as this determines in which column the tetromino can be dropped, as not all columns are always available. The branching factor also depends on the tetromino as some tetrominoes are symmetrical in one or two axes like the I and O shape, while others (the T tetromino) can be rotated in four unique ways (see figure 1.1). The combination of the large state space and large branching factor makes the traditional AI approach of a game tree unfeasible. Tetris is a hard NP-complete problem as shown by Demaine, Hohenberger, and Liben-Nowell (2003). This means an optimal policy is not effectively found by traditional methods and thus is machine learning a clear candidate to approach such a problem. Many games have simple rules and clear goals. This makes them ideal for RL, as performance is easily measured by the score. For Tetris this is also the case. The score can also be used as a reward for agents.

In this paper the effect of state representation in reinforcement learning applied to Tetris is examined. Different combinations of state representations were selected to find out what type of representation is most important. Agents for each combination were trained and their averaged scores compared over time. The hypothesis is that a very specific representation of the state without redundancy will yield the best results.

2 System Description

A simulation of Tetris was made so that it could be run at fast speeds and communicate easily with the rest of the system. This implementation was based

[2]https://tetris.com/bios

on that of Pavel Benáček [3] and can be found here[4]. This implementation features the seven tetrominoes found in the original game and features the same game logic. The only thing that is not implemented is the advanced technique called "tucking" where a block is moved at the last second so it can be moved underneath another block. This is because it is only a minor feature in the game that is hard to implement for a system such as this.

Every epoch (one game) starts with a new random tetromino is generated and placed at the top of the board. Then the state representation is generated for every legal option according to the game logic. Possible after states are generated by moving the tetromino to every horizontal position and dropping it down as far as possible. This is done for every possible rotation. This means that an L tetromino generates more after states than an O tetromino that only has unique one rotation. This process results in an array of representations that represent possible after states. This array is passed to the algorithm to make the decision of which move to make. The choice is made and a new tetromino is generated. This cycle is repeated until the game ends when the stacked tetrominoes reach the top of the board. After the game ends, the score is recorded. The game is reset and played again until the training epochs parameter is reached. This parameter was initially set to 1000, but after some exploratory runs no convergence was found. Therefore the number of training epochs was tripled to 3000. This seems enough for the average score to converge.

In Tetris and most other games, in order to score a point and to receive a reward, the agent must take many actions. This is a problem because how do you know how much and which action contributed to the reward? This was named the temporal credit assignment problem by Minsky (1961). If this sparsity becomes large, a lot of exploration is needed before a reward is found. If the rewards become too sparse, the reward is seldom found and learning never takes place. To circumvent this problem, it is assumed that the agent can always place the block where it wants and that their execution is flawless. This reduces the placing of one tetromino from many actions to one. In human games this is of course not the case. Then both the decision making and the placements are skills needed for a high score.

[3] https://github.com/benycze/python-tetris
[4] https://github.com/gjghendriks/TetrisRL

2.1 Temporal Difference learning

Algorithm 2.1 Temporal difference learning

Input: the policy π to be evaluated
The number of epochs

Where: $V()$ is the value function
S is some state
S^+ is the set of all states
A is an action
R is the reward given by taking action A.
α is the learning rate
γ is the discount rate
ϵ is given by equation 2.3

Initialize $V(S) = 0$, $\forall S \in S^+$
for all epochs **do**
  Initialize S as starting state
  while S is not terminal **do**
    if randomchance $< \epsilon$ **then**
      $A \leftarrow$ random action
    else
      $A \leftarrow$ action given by π for S
    end if
    Take action A, observe R, and next state S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  end while
end for

The decision algorithm is based on Temporal Difference (TD) learning algorithm outlined in Sutton (1988) and further discussed in Sutton and Barto (1998). The goal of RL is to achieve the maximum reward by constructing some policy. The policy dictates which action to take in which state. TD learning is a class of RL methods that try to approximate a value function, $V(S)$, that estimates the value of after states. This value is a measure of how much reward some after state will yield. Every iteration of TD learning, the value function is improved by updating it. Unlike Monte Carlo methods, TD learning tries to bootstrap itself based on the value function. The algorithm is an ϵ-greedy policy meaning that it will take the highest value

action most of the time, but sometimes take a random action depending on ϵ.

A reward of 1 is given for each cleared row. A negative reward of -10 is given to the agent when it dies to discourage it from doing so. After playing, the agent is able to predict the future rewards of some after state. This is done by the value function $V(S)$. This prediction gets updated every step and therefore is able to predict the value of some state better and better over time. The update function can be found in equation 2.1. The reward when the agent dies is given by equation 2.2.

$$V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)] \quad (2.1)$$

$$V(S) \leftarrow V(S) + [-10] \quad (2.2)$$

The action taken in some state is dictated by the highest expected reward of a state given by $V(S)$. This means that the algorithm may not score a point now, when it expects a higher reward later. This might mean making a seemingly bad move now, if this move results in the ability to make a very good move later. However, the expected rewards are multiplied by the discount rate (γ) (in this case 0.95) so that the immediate reward is valued more than later rewards. The pseudo code can be read in Algorithm 2.1.

Exploration is also an important part of TD learning. The exploration rate is given by formula 2.3 and depends on the current epoch T. This equation results in a logarithmic descent of the exploration rate (ϵ) in the range $[0.289, 0.031]$. This causes the system to take a random action some percentage of the time, but this percentage becomes smaller the more games are played. A declining exploration rate is standard in RL, as in the beginning the agent has little training and thus needs to seek out information more. While later in the training it can rely more on the knowledge already gained and performance is hurt by the random moves. This is similar to how humans learn tasks, as they try out strategies on new tasks, but use experience on known ones.

$$\epsilon = \frac{1}{log_{10}(T + 50)} - 0.3 \quad (2.3)$$

The policy relies on the value function $V(s)$ that makes a prediction of the expected reward for each state. The policy uses this value function to predict the value of future states. The value function is classically a big table where the expected value for each state-action pair is stored. However, because the state space of Tetris is so large, this is not feasible. Instead a function that returns the value depending on the state-action pair was approximated. This function was approximated by a multilayer perceptron (MLP). An MLP can, according to the universal approximation theorem, approximate such a function as described by Csáji (2001). The hyper parameters of the MLP that was used can be found in appendix A. The values were chosen on the recommendation of the supervisor of this paper.

2.2 State representation

The state representation of the game that was used always consisted of a 1D vector of proper fractions. These represent normalized values of the representation. This means that the maximum value was first calculated and that the value was divided by this maximum. The first ten numbers represent the column heights from left to right. Then more numbers may or may not follow depending on the tested condition. The options include:

- **Diff:** Nine numbers representing the difference in height between the columns from left to right.
- **Max:** A number representing the height of the highest column.
- **Holes:** A number representing the cells that are not filled but are covered by a tetromino and thus rows above need to be cleared first before they can be reached.
- **Wells:** Ten numbers representing the deepness of the well in that column if there is any in that column. A well exists when both neighboring columns have a bigger column height. When the column is neighboring the side of the board only the other neighbor is considered.

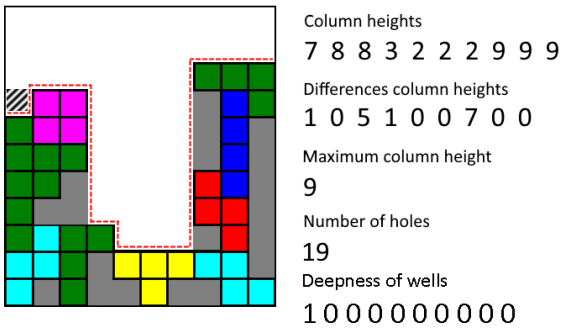

Figure 2.1: Schematic overview of representation of the board state. The grey squares indicate holes, the red-dotted line outlines the column height and the black and white striped square indicates a well. Note that a 10x11 board is pictured here, but the experiments are ran on a 10x20 board.

A schematic overview of the different representations can be found in figure 2.1. Not all 16 combinations of representations were tested, but the following were chosen:

- **Column Height** only
- **Column Height**, **Diff**
- **Column Height**, **Max**
- **Column Height**, **Holes**
- **Column Height**, **Wells**
- **Column Height**, **Holes**, **Wells**
- **Column Height**, **Diff**, **Max**, **Holes**, **Wells**

Thiery and Scherrer (2009) give an overview of Tetris controllers and their performance. They also outline the features that you can use. From this overview the most common features were chosen. For comparison a random agent was also used. This random agent always makes a random move.

2.3 Analysis

After each epoch the final score was recorded. Each configuration played 3000 games consecutively. This was repeated five times. The resulting data was then averaged over each configuration. A running average of 100 is then used to smoothen the plot as there is a lot of variance.

Hyperparameters were coarsely tuned to maximize the obtained scores for the **Diff, Max, Holes, Wells** representation. A flat exploration rate was investigated as well as a larger learning rate and a smaller discount rate. Much worse results were achieved and thus discarded.

3 Results

The results of the experiment that was described in the analysis section (2.3) is plotted in figure 3.1. We can see that the state representation with **Column height** and **Max** achieves the highest average score of around 23. The most specific representation with **Column Height**, **Diff**, **Max**, **Holes**, **Wells** also performs well, although it takes a bit longer to rise to a high performance. The other representations all have similar performances to each other. We can also see is that performance drops after a certain amount of epochs.

The random agent scored an average of 0.0156 points. Because the number is so small and does not change over time, this was not shown in the graph.

4 Discussion

4.1 Performance

Figure 3.1 shows that the **max** configuration is the best performing configuration. The **diff max holes wells** configuration also performs well. The other configurations clump together at the bottom. It seems that the **max** representation is crucial in the representation to achieve a good performance, as it is present in the two best performing configurations. Adding information to that does not seem to improve performance. The information that the **max** representation gives can be inferred from the **Column Heigth**. This makes it surprising that the **max** representation performs so well. A reason for this could be that the value function can more easily predict the value of some state when **max** is included because this encapsulates the value of the state (to some extent). Meaning that when **max** is low, the (true) value of the state is very likely to be high. Another, more general, reason could be that redundancy is very important in a system like

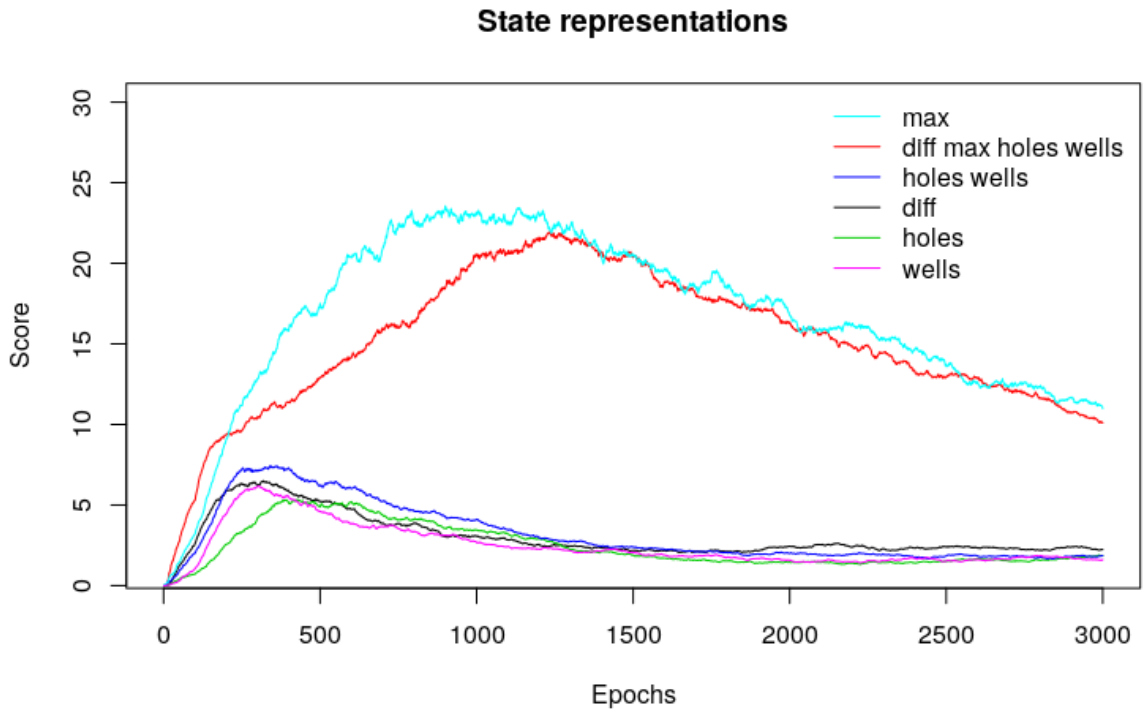

Figure 3.1: Running average of score over time for each state representation.

this. This is likely due to the instability of the system and in particular the MLP. The instability also might be the reason why the score declines after having reached its peak. This could be because of the declining exploration rate, however, the epochs after which the decline starts is different for the configurations.

In comparison to similar systems this system outperforms them or achieves similar results. Melax [5] showed significant learning but on a very simple version of Tetris. Carr (2005) realized a TD agent that achieved a performance of < 12 lines cleared for his implementation of the full Tetris game. Pirnay and Arabagi (2009) achieved an average of less than one line cleared using TD learning and a full state representation, meaning a vector of 200 numbers representing each cell. Thiam, Kessler, and Schwenker (2014) researched the influence of the exploration rate (ϵ) and found that for a flat value of 0.001 they where able to play 400 pieces per game after training. Unfortunately pieces per game are not directly translatable to lines cleared per game, but a rough estimate would be that, since all tetrominoes consist of four cells and a line consists of 10 cells, 2.5 tetrominoes per line are needed. However, for a game where you fill the board without clearing any lines, 45 tetrominoes can be still played. Because the board has 10x20 cells, of which one row (20 cells) remain unfilled, 180 cells can still be filled without clearing a line. Every tetromino is four cells, thus $180/4 = 45$ tetrominoes. This means that 400 pieces translates to at least $(400-45)/2.5 = 142$ lines cleared and at most $400/2.5 = 160$ lines cleared.

These results are still not up to par with an average human player, who can easily score a couple of dozen points. And certainly not the level of play Tetris champions like Joseph Saelee display, who can clear thousands of lines. This expert level performance can be achieved by RL systems as shown by Vinyals et al. (2019) for Starcraft II and the openAI team (2019) for Dota 2. These two examples

[5] http://melax.github.io/tetris/tetris.html

have millions of dollars of funding and months of training time however.

Other approaches to Tetris are able to perform even better. For example Groß, Friedland, and Schwenker (2008) applied RL to Tetris by using a weighted reward function. Their system was able to play for 20,000 tetrominoes on average. RL as a whole is not the best approach for solving Tetris as shown by Thierry and Scherrer (2010), who by using a cross-entropy method, were able to reach an average score of almost one million.

4.2 Random agent

The random agent got an average of 0.0156 points. This shows that the system, even when using the worst performing state representation, is able to make significantly more informed choices than random ones.

4.3 Training time

The training of the system over 3000 epochs took about 40 hours to complete on a laptop. For one epoch (one game) an estimated 40 - 400 tetrominoes must be placed. This depends on the score as a higher score results in more decisions and thus a longer training time. For every decision the MLP must be trained. Depending on the configuration, the MLP contains 60 - 90 nodes. This means that in the order of billion calculations must be done for only one configuration. This shows that one laptop and the time frame for this project are not sufficient for a project like this. To achieve expert level play, months of training on super computers is needed for complex games.

4.4 Further research

The most obvious next step in this research is to see what the performance is for every configuration of representations. Especially interesting would be the other configurations with **max** included. Maybe one could perform even better than the **column height** and **max** configuration that was tested. It would also be interesting to consider removing the column height to see if only the information provided by **max** could be enough to perform well. This could give more insight into what each representation contributes.

New representations could also be thought of. For example: the total amount of wells, average distance to top and the holes per column instead of a total holes of the field.

Of course it is also interesting to know what would happen when the system is ran for even longer. Do the configurations all converge or do they increase in performance again? What we do know from the openAI team (2019) is that to achieve great results, extreme training time is needed. They achieved great results, but had to train their system on supercomputers for 10 months.

TD learning is super dependent on hyperparameters, in particular exploration rate (ϵ) and learning rate (α). In this paper they where chosen from experience of the supervisor and tuned by hand after that. The chosen values seemed to work. However, as always when using this kind of system, especially when using an MLP, the tuning of these parameters is a hard problem to solve. A parameter search could be done where a value is chosen by the system automatically and the performance tested. Then the next value is chosen and so on. This requires a lot of time as for each parameter value the system must first be trained before performance can be recorded. The problem worsens for each additional parameter that is tuned this way.

5 Conclusion

This thesis set out to research the effect of state representation in reinforcement learning applied to Tetris. It was found that the difference in performance is significantly dependent on the state representation. It is also found that redundancy in state representation is important in RL systems. The best configuration of state representation achieved an average score of 23.

References

David Andre and Stuart J Russell. State abstraction for programmable reinforcement learning agents. In *AAAI/IAAI*, pages 119–125, 2002.

Murray Campbell, A. Joseph Hoane, and Fenghsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1–2):57–83, January 2002. ISSN 0004-

3702. doi: 10.1016/S0004-3702(01)00129-1. URL shorturl.at/giluY.

Donald Carr. Adapting reinforcement learning to Tetris. Master's thesis, Rhodes University, dec 2005.

Balázs Csanád Csáji. Approximation with artificial neural networks. Master's thesis, Eötvös Loránd University, 2001.

Erik D. Demaine, Susan Hohenberger, and David Liben-Nowell. Tetris is hard, even to approximate. In Tandy Warnow and Binhai Zhu, editors, *Computing and Combinatorics*, pages 351–363, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45071-9.

Thomas G Dietterich. State abstraction in MAXQ hierarchical reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 994–1000, 2000.

Alexander Groß, Jan Friedland, and Friedhelm Schwenker. Learning to play Tetris applying reinforcement learning methods. In *ESANN*, 2008.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

M. Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961.

Hans Pirnay and Slava Arabagi. Temporal difference learning in the Tetris game. Technical report, RWTH Aachen University, 2009.

Tim Salimans and Richard Chen. Learning montezuma's revenge from a single demonstration. *ArXiv*, abs/1812.03381, 2018.

David Silver et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016. ISSN 1476-4687. doi: 10.1038/nature16961. URL https://doi.org/10.1038/nature16961.

Richard S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44, 08 1988. doi: 10.1007/BF00115009.

Richard S. Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*, chapter 6, pages 115–132. MIT Press, 1998.

Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 7 1994.

the openAI team. Dota 2 with large scale deep reinforcement learning. *arXiv*, abs/1912.06680, 2019.

Patrick Thiam, Viktor Kessler, and Friedhelm Schwenker. A reinforcement learning algorithm to train a tetris playing agent. In Neamat El Gayar, Friedhelm Schwenker, and Cheng Suen, editors, *Artificial Neural Networks in Pattern Recognition*, pages 165–170, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11656-3.

Christophe Thierry and Bruno Scherrer. Improvements on learning Tetris with cross-entropy. *International Computer Games Association Journal*, 32(1):23–33, March 2010. URL shorturl.at/oqsSV.

Christophe Thiery and Bruno Scherrer. Building Controllers for Tetris. *International Computer Games Association Journal*, 32:3–11, 2009. URL https://hal.inria.fr/inria-00418954.

Oriol Vinyals et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, Nov 2019. ISSN 1476-4687. doi: 10.1038/s41586-019-1724-z. URL https://doi.org/10.1038/s41586-019-1724-z.

A Appendix: Hyper parameters

The MLP has three fully connected layers. The input layer shape depends on the state representation but is at least a size of ten as the **Column Height** is always included in the representation. The optimizer that was used is the Adam optimizer as described by Kingma and Ba (2014). The loss function used is the mean squared error function.

Parameter	Value
Learning rate (α)	0.0001
Discount rate (γ)	0.95
Training epochs	3000

Layer	Nodes	Activation function
Input	depends	none
Hidden	50	sigmoid
Output	1	linear