Final Report - Drop7

Erez Klein
Senior Algorithms Engineer - Nvidia
Tel Aviv, Israel
erezk@nvidia.com

Ben Friedmann
Senior Algorithms Engineer - Nvidia
Haifa, Israel
benfr@nvidia.com

1 INTRODUCTION

The goal of this project is to design an artificial intelligent agent that can achieve human like performance in playing the game of "Drop7". Drop7 is a simple puzzle game with easy to comprehend rules (see section [1.1] below), however it has a high level of sophistication with an enormous number of possible states ($\sim 10^{49}$). On top of that, the game have a significant amount of randomness incorporated in it. The stochastic nature of the game with its vast state-space, rules out the possibility of a closed form solution approach.

In general, there are three types of machine learning; supervised learning, unsupervised learning and reinforcement learning [11]. The later, uses an iterative approach to gather feedback from its environment, finding the most effective route of action. Given the large state space we decided reinforcement learning is the most suitable technique to face this project's challenges.

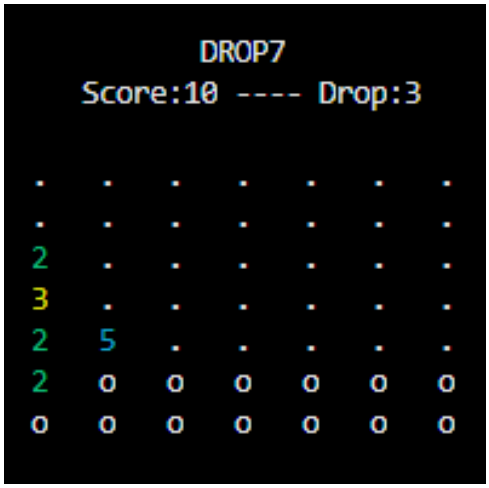

Figure 1: Drop7 Simulator

1.1 Game Rules

The game is played on a 7x7 square grid (Figure 1). In each turn the player is assigned a disc with a random number from 1 to 7 ('number disc' / 'element') and is given an option to drop the disc on any of the possible columns (7 possible actions). Whenever the number of a disc matches the number of the contiguous discs in its row or column, that disc explodes, effecting any blank discs in its immediate proximity (up, down, left, right). When a disc explodes near a blank disc once, it gets cracked, and when a disc explodes near blank-cracked disc, it turns into a random numbered disc. Thus, a blank disc is converted into a number disc only after a disc explodes near it twice. After every 5 turns, the round ends and a full row of blank discs emerges from the bottom of the grid, pushing all other discs up. Each turn the player gains 1 score point. The objective is to eliminate discs for as long as possible, accumulating score points, until the grid overflows and the game ends.

2 LITERATURE REVIEW

In our literature review we have not found works that tried to solve Drop7, thus we took interest in related works of reinforcement learning algorithms that play different games. Most of the literature reviewed was focused on deep reinforcement learning [2] [8] [12] [10], which given the time constrains of this project was not a feasible avenue to pursue. Given the size of possible states in the problem, and given the fact we are using linear function approximation we continue with the understanding that our success will be limited. Instead, we studied the subject of function approximation in Q learning. We took notice of Samuel, Arthur L work [9] on what features to take and the possibilities and limitations of choosing the correct features. However we did differ by the method of updating our weights. In Samuel's work he ignored the value of the terminal states that he reached, we on the other hand used the observed rewards that we got (the score of the game).

3 DATASET

Typically when using reinforcement learning methods, the algorithm does not train on a dataset per se, but rather an agent explores the environment creating a dataset on the go. In our problem we use a similar approach. In the training stage, the dataset is comprised from the Q-Learning algorithm playing the game many iterations, recording the score and actions and updating the features' weights accordingly. In order to easily extract interesting information of a certain state, a game class was built to hold more information then the raw $7x7$ matrix. Analysis of the board's various data structures were created such as the amount of detonations that occurred, the top free location of each column, and the groups of elements in each row. This extra space and analysis is done during each round and helps to create more meaningful feature extractions in addition of simplifying code execution and reducing the chance for software bugs.

Once the training stage is done, the test stage begins, relying on the weights that were found during the training stage.

4 BASELINE

The baseline was obtained by a random policy. Namely, a random agent played the game 5,000 times, in each turn following a random policy and choosing one of the 7 available actions uniformly $c \sim U(1, 7)$. At the end of each game, the final score was logged (Figure 2). Both the average and the standard deviation were calculated

to determine the overall baseline:

$$\textbf{Baseline} = \{\text{Avg} : 31.2,\ \text{Std} : 5.12\}$$

First, it is worth noting that achieving a score of ~ 31 is not always guaranteed (although likely) even for a human player. Second, note that the random agent's best score was almost as high as the human player's average (oracle) (Figure 2 and Figure 4). Both of these points emphasize even more the complexity of the game induced by its stochastic nature, and that the luck of the draw, both in the number elements that are introduced from the blank discs and the element the player is given to drop, is an important aspect of one's ability to get a high score. A baseline of pure random might seem a bit too relaxed in first glimpse, but in fact is far from trivial.

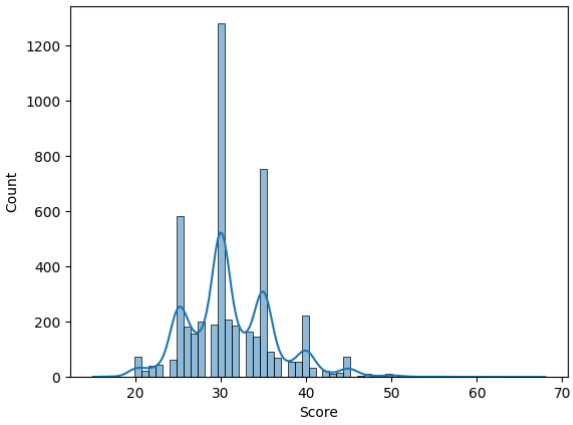

Figure 2: Random approach - histogram of high scores

5 MAIN APPROACH

The main approach that was taken, as mentioned previously, is reinforcement learning. To achieve high performance in the game, one must develop a winning strategy. For an artificial intelligent agent, that means to learn the best policy it can find. To formalize this problem the environment in which the agent acts must be well defined - a model of Drop7 is needed. A Markov Decision Process (MDP) is the natural choice for such a model:

- **State**: $s = (\mathbf{M}, n)$, where $\mathbf{M}$ is a $7x7$ matrix, representing the current board and each cell value. n is the current disc number ('element', 1-7) the player is required to drop next.
- $s_{start} = (\bar{\mathbf{0}}, \mathbf{n}),\ \mathbf{n} \in [\mathbf{1}, 2...7],\ \bar{\mathbf{0}} \in \mathbb{R}^{\mathbf{7x7}}$. Note that there are 7 possible start states, all with probability of $\mathrm{p}(n = i) = \frac{1}{7}$.
- **Actions(s)**: $\{a = \text{col} \mid \text{col} \in [1, 2, 3, 4, 5, 6, 7]\}$. The player can drop the current element at any of the 7 columns of the grid.
- **T(s,a,s')**: probability of s' if action a is taken in state s. This probability is quite cumbersome to calculate, duo to the enormous number of possible successors. Even s_{start} with action $a = i$ has 7 successors, depending on the next element that arrives. Hence, after only one turn there are already $|s_{start}| \cdot |\text{possible actions}| \cdot |\text{possible next element}| = 7^3$ possible successors states.
- **Reward**$(s, a, s') = \mathbf{1}[s' \neq s_{end}]$. Each move will grant the player 1 score point, as long as the game did not end.
- **IsEnd**(s): if $\mathbf{M}$ is overflowed in at least one column.
- **Discount factor**: $\gamma = 1$.

Although well defined, the model above suffers from a number of difficulties. It has an extensive amount of valid states ($\sim 10^{49}$ which is approximately the number of atoms on Earth!). Furthermore, calculating the transition probabilities **T(s,a,s')** for each (s, a, s') is theoretically possible but in practice not feasible. Nevertheless, now the agent can start the tedious process of policy optimization. First, to get to know the environment, the agent must be able to explore. Furthermore, the agent needs to take advantage of the already purchased knowledge to improve its performance. Hence an epsilon-greedy approach was selected, balancing between exploration and exploitation [11]:

$$\pi_{act}(s) = \begin{cases} \textbf{argmax}_{a \in \textbf{Actions}}\ \hat{Q}_{opt}(s, a), & w.p\ \ 1 - \epsilon \\ \textbf{random from Actions}(s), & w.p\ \ \epsilon \end{cases}$$

Where ϵ is an hyper parameter (see section 5.1).
Secondly, an approximation of $\hat{Q}_{opt}(s, a)$ is in order. To accomplish that, Q-learning algorithm was applied [14][11][1]:

$$\hat{Q}_{opt}(s, a) \leftarrow (1 - \eta)\hat{Q}_{opt}(s, a) + \eta[r + \gamma \max_{a' \in \textbf{Actions}(s')} \hat{Q}_{opt}(s', a')]$$

Where $\gamma = 1$, and η is an hyper parameter (see section 5.1).

However, due to the number of possible states, the algorithm is not capable of covering every option. In practice, the agent's probability to encounter the exact same state in two different games is slim [3].
In order to address this issue, function approximation was introduced to the Q-learning algorithm [14], based on feature vector $\phi(s, a)$ and weight vector $\mathbf{w}$:

$$\hat{Q}_{opt}(s, a; \mathbf{w}) = \mathbf{w} \cdot \phi(s, a)$$

Where $\mathbf{w}$ is learned in an iterative manner:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\hat{Q}_{opt}(s, a; \mathbf{w}) - (r + \gamma \max_{a' \in \textbf{Actions}(s')} \hat{Q}_{opt}(s', a')]\phi(s, a)$$

$$\mathbf{w}^{(0)} = \bar{0}$$

This practice, with well defined feature vectors $\phi(s, a)$, will reduce the state space substantially. The features should maintain a delicate balance between dimension reduction and holding as much relevant information as possible. Many features were evaluated in an attempt to refine and enhance the agent's performance [4]. Finally, the following features were introduced:

- **'min_eq_elem'**: $\mathbf{1}$[min_col == a], equals to 1 when the element is dropped on one of the lowest columns. Range $[0 : 1]$.
- **'max_eq_elem'**: $\mathbf{1}[max\_col == a] \cdot \mathbf{1}[|max\_col| \leq 2]$, equals to 1 when the element is dropped on one of the highest columns, in case there are no more than 2 columns with the same maximum value of elements. Range $[0 : 1]$.
- **'row_dets'**: $\sum_{i=1}^{7} \mathbf{1}[\mathbf{M}(row(elem), i)$ detonated$]$, equals to the sum of discs in the current element's row that were detonated. Range $[0 : 6]$.

- **'col_dets'**: $\sum_{j=1}^{7} \mathbf{1}[\mathbf{M}(j, col(elem))$ detonated], equals to the sum of discs in the current element's column that were detonated. Range $[0:6]$.
- **'elem_det'**: $\mathbf{1}$[current element detonated]+**det_near_disc()** (see below), equals to 1 if the current element detonated, and gets a bonus if its next to black discs. Range $[0:7]$.
- **'1_det'**: $\mathbf{1}$[current element == 1 detonated]+**det_near_disc()** (see below), equals to 1 if the current element is a disc number '1', and its detonated. Gets a bonus if its next to black discs. Range $[0:7]$.
  - helper function: **'det_near_disc()'**: $\sum_{i=d,l,r} \mathbf{1}$[detonated]$\cdot(\mathbf{1}$[i == blank]$+2\cdot\mathbf{1}$[i == cracked]), that is, if the current element detonated, any of its neighbors (down, left, right) that are blank discs will increase the feature value by 1, and each neighbor that is a cracked disc will increases the value by 2. Range $[0:6]$.

Even though the function approximation solves many issues, like the model's dimension, it introduces new limitations. The function approximation is linear, that is, the different features do not interact. This holds an hidden assumption that the features are independent, which they are not, and their weighted superposition can dictate the best course of action. This is a very strong assumption, that holds only in case the features where chosen in such a way that the state-space under this transformation is linearly separable. Because this is almost certain not to be the case, most of the selected features do not reflect the status of the game as is, but rather the different aspects of the outcome following a certain action. This allows the agent to understand what are the effect of its actions, learning which states are more favorable.
However, now the model faced a new problem - overfitting. As many weights-based models do, this liner function approximation model suffers from overfitting when the training stage is too long. To handle this issue the Ridge $\lambda\|\mathbf{w}\|_2^2$ regularization was added, modifying the weights update as such:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta\left\{[\hat{Q}_{opt}(s,a;\mathbf{w}) - (r+\gamma \max_{a'} \hat{Q}_{opt}(s',a')]\phi(s,a) + \lambda\cdot\mathbf{w}\right\}$$

Preventing the model from overfitting, where λ is a hyper-parameter (see section 5.1)
Be that as it may, equipped with the modified algorithm and the above features, the agent can try and tackle the unfamiliar environment, its target; achieving the highest score. Although the feature sate space is smaller in orders of magnitude from the original problem, it still takes a while to train on. Thus, the agent played $n = 50,000$ games (Figure 3). Next, as a evaluation stage, 10,000 games were played, this time with $\epsilon = 0$ (Figure 4), that is - no exploration, only exploitation. The average score combined with its standard deviation were used to assess its success level.

5.1 Hyper Parameters Tuning

As in any machine learning model, there are hyper-parameters that need to be tuned in this model as well. Each has its own effect on the training process of the agent, and therefore on the final result. Tuning those parameters can be cumbersome work, done by combination of past work, intuition and many iterations of trial and error. Here are some of the hyper-parameters of this model:

- ϵ, the probability of exploration in the greedy-epsilon algorithm: Although the agent was able to achieve fair performances with a constant ϵ (0.2), finally a more robust approach was taken to fully utilize the exploration-exploitation tradeoff. Accordingly, ϵ was set to:

  $$\epsilon = \frac{1}{\sqrt[4]{i}}, \quad i = \text{iteration number}$$

  The value updates on each iteration, decreases very slowly, reaching a value of 0.1 after 10,000 iteration. This allows the agent an extended period to explore while still in the training stage.
- η, the gradient step size of the weights: Initially this parameter was set to $\frac{1}{\sqrt{i}}$, however from an algorithmic stability point-of-view, it is recommended that η will satisfy the following [6]:

  $$\sum_{i=0}^{\infty} \eta_i = \infty \quad \& \quad \sum_{i=0}^{\infty} \eta_i^2 \leq \infty$$

  Hence, η was chosen to be:

  $$\eta = \frac{1}{i}, \quad i = \text{iteration number}$$

- λ, regularization factor: Many different values were tested in order to reach convergence on one hand, and yet have a long enough learning period on the other. The value that provided the best outcome is:

  $$\lambda = 0.1$$

6 EVALUATION METRIC

Generally, the evaluation metric is based on the game's score. The score of the game is equivalent to the number of turns played before the grid overflows. Accordingly, the performance of a player (human or an artificial agent) is measured by:

(1) **Avg**: average score of all the played games.
(2) **Std**: standard deviation of all played games score.

The two metrics grant a clear overview of how good, as well as how consistent a player is. For an agent that has both training and test stages, like the reinforcement learning agent, the evaluation metric is defined as the measured {Avg, Std} on the test stage, that is $\epsilon = 0$.
In the training stage for the reinforcement learning agent, additional more "amorphic" metrics were used to fine tune the model's hyper-parameters, particularly which features to incorporate into $\phi(s,a)$. For instance, the slop of the score graph as a function of games played (Figure 3). A Steeper slop indicates a higher gradient, that might indicate not only a faster learning rate but maybe also a higher asymptotic value. Another example is the feature's weight updates. Final weights that remained very small might indicate that the correlating feature is not contributing much, and the agent de facto ignores them. For weights that did not converge and jitter strongly even after a substantial number of updates, the correlating features were discarded, assuming they were only increasing the variance while not benefiting the average final score.

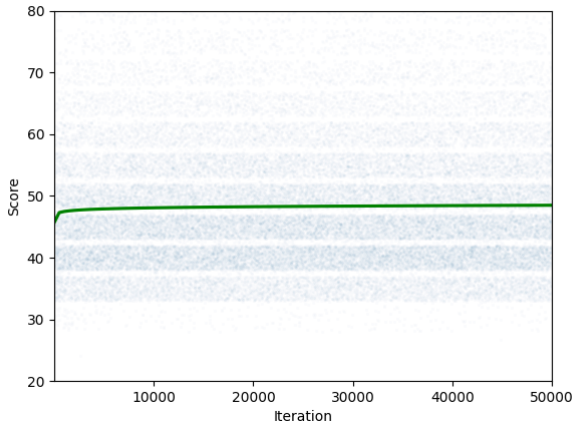

Figure 3: Q-learning progress as a function of the current iteration. The green line is a logarithmic fit, and each blue dot is a score that the RL algorithm has reached.

7 RESULTS ANALYSIS

The reinforcement learning agent trained by playing Drop7 50,000 iterations (Figure 3), and than was put to a test; playing 10,000 games on policy, that is with $\epsilon = 0$. Each turn taking the best move its newly learned policy dictate. The final results that were logged are the following (Figure 4):

$$\textbf{RL Agent} = \{\text{Avg} : 49.61,\ \text{Std} : 11.18\}$$

Figure 4 is comparing the reinforcement learning agent performance with the random agent (baseline) which achieved $\{\text{Avg} : 31.2,\ \text{Std} : 5.12\}$ on 5,000 games, and the humans accumulating score is (oracle): $\{\text{Avg} : 73.2,\ \text{Std} : 21.4\}$ on 30 games.

As one can see, the reinforcement learning agent surpasses the random agent, but falls short comparing to a human player. For better intuition, transforming the result to a scale of 1 to 10, where the baseline is ranked as 1, and the oracle as 10, the agent will get a rank of ~ 5.5.

Note that non-negligible number of games, the agent was able to attain a better than human average score. The 'Std' metric of the reinforcement learning agent is fairly high, but may be misleading. The the final score distribution is not a symmetric Gaussian, but rather has a heavier tail to the positive side of the X axis, indicating that the agent can preform much better (doubling its score) on certain games (Figure 4).

Notwithstanding the fact that the agents score is way lower than the oracle, based on the simple model of a one layer Q-learning algorithm with a function approximation, these results surpassed our expectations. One cannot determine if the current model exhausted its capabilities, but there is evidence that the model is probably not too far from its edge, performance-wise. Most of the sophistication of the model is rooted in its features, that were 'filtered' by the model designed statistic point of views. Enhancing this model, by tweaking and modifying it, might result in even better performance, but from a reinforcement learning stand point is quite senseless. Hence, to realize the upper target threshold (human-like skills), a new model is in order.

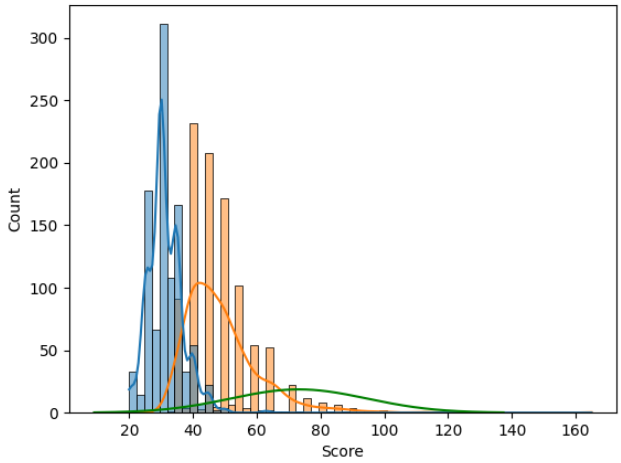

Figure 4: A comparison between human performance (Green), RL performance (Orange) and random performance (Blue).

8 ERROR ANALYSIS

As the number of features grew, and due to the enormous sate space of the problem, many experiments comparing different training stages lengths were done, trying to evaluate the marginal additional benefit from a longer training sequence. The underlining assumption; longer training - more optimal policy. To our surprise, this was not the case.

With the improved model, tuned hyper parameters and carefully chosen features, the agent got promising results with a small size of iterations, but got worst and worst with more and more games. Moreover, the features' weight values skyrocketed. Digging into the problem, the conclusion was that the root cause is a mixture of two effects:

- No guarantee of convergence: this model uses bootstrapping, function approximation, and is an off-policy model. The combination off these three attributes into one model, can cause the model to be unstable and to diverge.
- Overfitting: the model is linear in the feature's weights, while the features (a) decrease the state-space significantly, and (b) some features are strongly correlated. The combination of these qualities will make the model prone to overfit.

To overcome the new issue, ridge regularization was introduced to the model. A new component was added to the weights updated:

$$\frac{\partial(\frac{1}{2}\lambda\|\mathbf{w}\|_2^2)}{\partial\mathbf{w}} = \lambda\mathbf{w}$$

The modified model has regain its stability, and the feature's weights converged to finite reasonable values. Figure 5 compares the exact same model at the test stage (10,000 games) posterior training it on 50,000 games, once with and once without the ridge regularization.

Without regularization the model's average score is lower than the baseline, preforming worst them random. With the regularization, the agent is able to archive worthy results, following a rational policy.

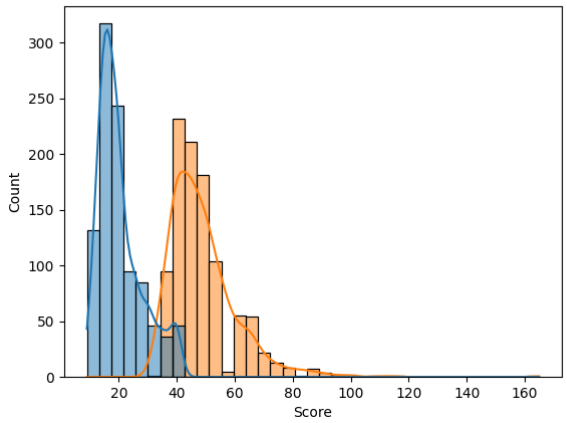

Figure 5: A comparison between two RL models, without regularization (Blue) and with $L2$ regularization (Orange).

9 FUTURE WORK

Although the agent's best scores are as good as the human's best scores, the variance is very high and the performance is not consistent. As discussed above, the current model is somewhat limited. Consequently, a better model is required [7]. One possible model is a deep fully-connected network that is essentially a cascade of Q-learning with evaluation function layers. Using a back propagation algorithm, all the weights and biases can be updated. This model will allow much more complex interaction between different features, hopefully enabling the agent to better preform [5][13]. Nevertheless, a larger model with more variables to tune will demand much more training, hyper-parameter fitting, and probably less stability. One limiting factor might be runtime, another might be hardware.

Having said that, we do believe that there is more to learn here, both for the agent and for ourselves, so we are planning to continue to more advance algorithms in the near future.

10 CODE

The code is uploaded to https://github.com/ekreate/cs221-final-project.

REFERENCES

[1] Leemon Baird. 1995. Residual algorithms: Reinforcement learning with function approximation. (1995), 30–37.

[2] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. 2002. Deep blue. *Artificial intelligence* 134, 1-2 (2002), 57–83.

[3] Tommi Jaakkola, Satinder P Singh, and Michael I Jordan. 1995. Reinforcement learning algorithm for partially observable Markov decision problems. *Advances in neural information processing systems* (1995), 345–352.

[4] Jundong Li, Kewei Cheng, Suhang Wang, Fred Morstatter, Robert P Trevino, Jiliang Tang, and Huan Liu. 2017. Feature selection: A data perspective. *ACM Computing Surveys (CSUR)* 50, 6 (2017), 1–45.

[5] Nicholas Lundgaard and Brian McKee. 2006. Reinforcement learning and neural networks for tetris. *Technical report, Technical Report, University of Oklahoma* (2006).

[6] Francisco S Melo and M Isabel Ribeiro. 2007. Convergence of Q-learning with linear function approximation. In *2007 European Control Conference (ECC)*. IEEE, 2671–2678.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.

[9] Arthur L Samuel. 1959. Some studies in machine learning using the game of checkers. *IBM Journal of research and development* 3, 3 (1959), 210–229.

[10] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

[11] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[12] István Szita and András Lörincz. 2006. Learning Tetris using the noisy cross-entropy method. *Neural computation* 18, 12 (2006), 2936–2941.

[13] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30.

[14] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.