Intelligent Agent for Game 2048 with Deep Q-learning

Kaiyuan Hou [*1] **Xuan Zhao** [*2] **Chenjie Deng** [*2]

Abstract

In this project, we have developed an intelligent agent for game 2048. We used many different ways, Monte Carlo Tree Search, deep Q-learning network and proximal policy optimization to train the agent. Our target is to make the agent play the game with as many steps as possible. We also compared our modified environment against the original environment to see how the reward function affects the rate of convergence.

Keywords: *game agent, reinforcement learning, 2048, deep q-learning, mcts*

1. Introduction

Games are not only entertainment, though. Training a virtual agent to outperform human players can help people know how to optimize different processes in a variety of different and exciting sub-fields.AI agents can be used in many aspects of our lives, such as autonomous driving, medical treatment, stock, military, etc.

In this work, we will introduce how to design an agent to play game 2048. The target of the game is to achieve as higher value as possible. 2048 is a famous single-player stochastic game since since 2014. In a game board with size of 4×4, user can move all the existing tiles on board to four directions: right, left, up and down and a random new tile with value (either 2 or 4) will spawn in the empty space on the board. If two tiles are adjacent and contain the same value, they will be merged into one single tile. The value of the new tile is the sum of values of the merged tiles. Each newly generated tile cannot merge with another tile in the same turn [1]. The games ends when there is not valid

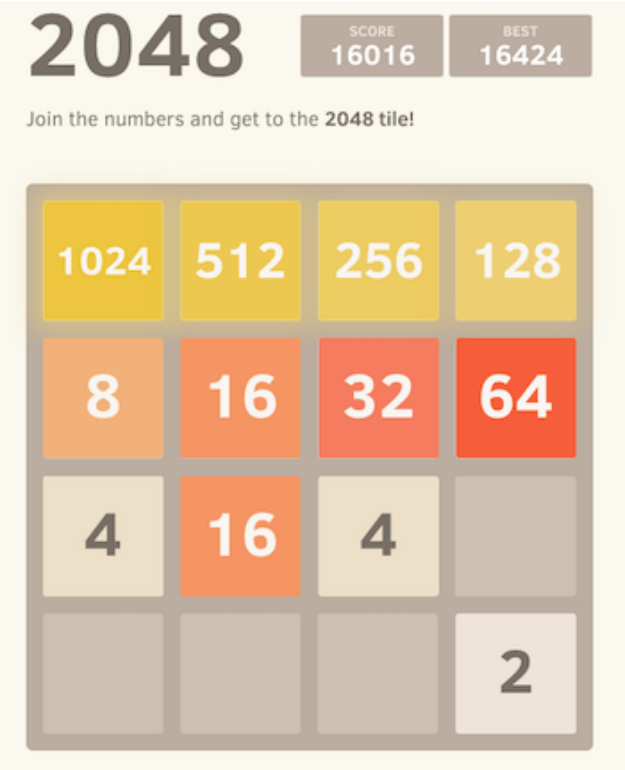


Figure 1. Game Interface

action can be performed and the target is to maximize the score. Every time two tiles are merged together to generate a new tile, the score is incremented by the same amount of the value on the new spawned tile which is the sum of value on two merged tiles.

Similar to games such as Go and chess, 2048 proves an interesting challenge for AI to play well since it has a very large state space. Even though the action space is discrete and only consists of four actions, the state space grows up exponentially with the size of board. In the board of 4×40, if we limit the value of tile to have a maximum of 2048. Each tile then can be either empty, or contain a 2, 4, 8, ..., 1024 or 2048 tile. There are 12 possibilities per tile and 16 tiles on the board, so there are $16^{12} =\approx 2.81 * 10^{48}$ possible board states. It would be infeasible to visit all the states to maximize the value function in each state.

[*]Equal contribution [1]Department of Electrical Engineering, Columbia University, New York, US [2]Department of Computer Science, Columbia University, New York, USA. Correspondence to: Chong Li <cl3607@columbia.edu>, Sam Fieldman <sf3043@columbia.edu>.

Proceedings of the 37^{th} International Conference on Machine Learning, Online, PMLR 119, 2020.

[1]https://en.wikipedia.org/wiki/2048_video_game)

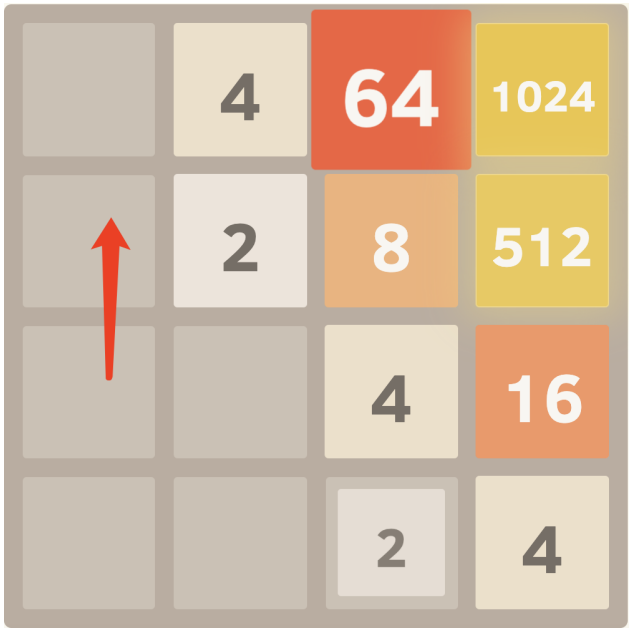


Figure 2. An example of illegal action.

2. Related Work

The idea of controlling an agent to find the optimal decisions when facing some uncertainties has gained increasing traction in the past decade. As early as in last century, (Tesauro, 1995) used Temporal Difference Learning algorithm achieved a master level of playing backgammon. Recent year, Google's DeepMind developed multiple AI agent for different games, such as playing Atari(Mnih et al., 2013), and the famous Go agents: AlphaGo, AlphaGo Zero and AlphaZero(Silver, 2016; 2017; 2018) that have the ability to defeat the world Go champion. Since 2048 is released recent years which is an age of using AI to play games (Mnih et al., 2015), it is natural that there have already been many AI commitments for this game. The design methods various a lot from simple maximizing expectation all the way down to complex reinforcement learning.

In terms of the agent developed by reinforcement learning on 2048, most of work we found, such as (Rui, 2015) and (mmcenta, 2021) are prioritizing the final convergence. In their work, the agent is able to tackle the game after the agent has been trained for 14 days. Since we don't enough time for tuning the parameters, we start will an idea that is to make the actual "learning rate" faster.

3. Design

In order to test the convergence rate of different designs, we make shrink size of board from 4 x 4 in the original game to a board of 3 x 3. In a 3 x 3 board, achieving a tile with value of 256 is about the same difficulty of achieving a tile with value of 2048 in a 4 x 4. If we assume that the highest tile can be achieved at every position of the board, then the total possible observation in a 3 x 3 board is $9^9 = 3.87 * 10^8$ while in a 4 x 4 board is about $2.81 * 10^{14}$ possible states. Because of the significant reduce in the observation space, we can have more time on tuning the models.

3.1. Open AI gym environment

3.1.1. Illegal move

Unlike other Open AI environment, the environment of 2048 didn't specify the end of game. In the real game, the game ends when there is not legal action. The illegal action is trying to move to one direction while the environment doesn't change after. An example illegal action is shown in Figure 2.If we are trying to move upward, the board will not change since there are not tiles can be merged. In this situation, the legal action can be either move to left or move downward. One way to avoid the illegal action is to choose the action from valid actions. But in this way, we violate the idea that let the agent learn to play by itself. So we chose a similar idea with (Rui, 2015). In (Rui, 2015), the environment has a fixed constraint for the maximum number of illegal move the agent can take. But we modify the fixed constraint to a dynamic number. In each game, we allow the agent to choose illegal move will a reLU like function.

$$\epsilon = max(10, \frac{k}{10}) \tag{1}$$

where ϵ is the number of allowed illegal moves and k is number of turns of the game. Compared to instantly ending the game after a single invalid action, these methods allow agents to recover from mistakes, such as an exploration step that resulted in no changes and explore more states.

3.1.2. Reward function

After we defined the terminal state, we need to think about the reward function, which is the most import part in reinforcement learning (Knox & Stone, 2012).

In the actual game human played, people are trying to maximize the overall value as stated earlier. We used similar idea here that is to maximize the score of game. That is, the reward function of each step is the difference between the score in new state and the score in the old state.

Initially, we define the reward in the OpenAI gym like the method motioned above. However we found that, with this reward function, the agent learns extremely slow. This is due to the factor that the agent usually chooses a illegal action. If an illegal action is performed, the agent have little learning significance and it can also pollute the reply memory. Moreover, when using off-policy algorithms, the

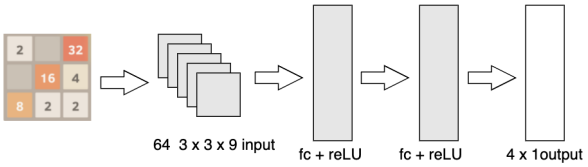


Figure 3. Neural network for deep q-learning network

agent may get stuck in a long sequence of those illegal action since the policy will always give the same action under that state.

As a result, we add an negative reward on performing an illegal action to penalize illegal actions. Besides the penalty on illegal actions, we also used some heuristics to the agent to increase the rate of convergence (Matignon et al., 2006). If the highest tile is on the corner, we offer some bonus reward. We know that if the highest tile is not at the corner, it has a higher tendency to loss the game soon.

3.2. Monte Carlo Tree Search

At beginning we think for the deep reinforcement learning problem, we need to find a correct estimate of the policy network target value. The main problem for DQN agent is that it converges very slow, due to the huge number of states ($\sim 10^{10}$) and game length (~ 60) for even the 3*3 version of the game. MCTS can solve this problem perfectly because with a search depth close to the game length, every episode of MCTS generates an unbiased estimation of the optimal state value. In other words, if we use a variable step length, the MCTS would converge exponentially fast.

We define the score of each state the average number of steps one could achieve when playing a 2048 game starting from that state. Note that the average is over all random policies and all possible state transitions. The goal of MCTS is to quickly estimate this score for a large number of states (ideally $\sim 10\%$ of the total number of states). We can then train the DQN policy network using this estimation as target. Therefore, we make use of an s-value table to store these estimations while we perform the MCTS. Another advantage of using the memory table is that we could give a more accurate estimation of state value of the last state in a trial, if the last state is not an end state, which could be very possible as the game length of an MCTS trial can always exceed our search depth (see Figure 5(a)). We could also make use of variable step length to give more accurate estimation with memory table. In our experiments, we use the global average (step length $\frac{1}{N}$), which makes sense since every trial is independent.

We did not expand the search tree towards more promising

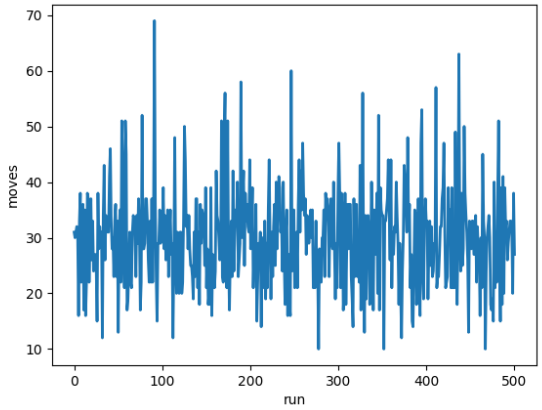


Figure 4. An agent with random policy in the 3 x 3 environment

children, and the performance of the algorithm is negatively affected because of this. Nevertheless, even with this simpler MCTS model we achieved an average game length of ~ 110 steps, see Figure 5(a), which is close to top-level human players like us and is far more than our baseline DQN agent in the early experiment.

As we need to train the policy network and update with back propagation. We started with a Monte Carlo Tree Search (MCTS (Browne et al., 2012)) algorithm to approach the correct policy. The MCTS look ahead of observation space with the current action selected and the find the best action to take at the current state. To increase the search efficiency, we used the multi-threading to process the tree search. With a search width of 20 and depth of 35, the tree search can be done in 3 seconds. (Coulom, 2007)

3.3. Deep Q-learning Network (DQN)

Since the observation space in this game is fully discrete. We implemented a deep q-learning network to learning this environment. The structure of the network is straightforward. It takes the observation which is presented by a 3 by 3 matrix as input. And put it to two fully connected layers and output a 4 by 1 vector which represents the 4 possible actions respectively.

highest tile	percentage
128	64.7%
256	17.3%

Table 1. DQN performance on a 3 x 3 board

With 40 hours trained, the agent is able to get a tile of 128

and 256 frequently. In table 1, we can see that the agent is able to get an 128 tile with a probability of 64.7% and an 256 with a probability 17.3%.

3.4. Proximal Policy Optimization (PPO)

Algorithm 1 PPO, Actor-Critic Style

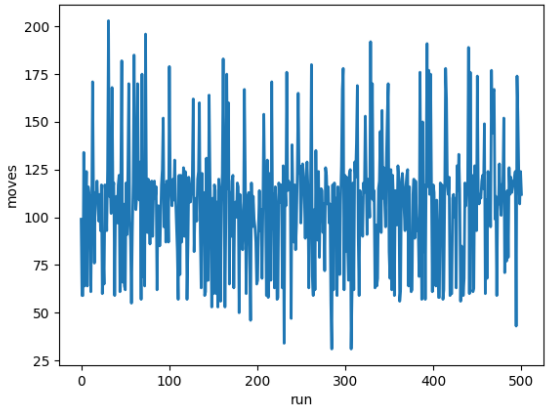
for $iteration = 1, 2, ...$ **do**
  for $actor = 1, 2, ..., N$ **do**
    policy $\pi_{\theta_{old}}$ in enviroment for T timesteps
    Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and mini-batch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$
end for

We also trained a proximal policy optimization (PPO) agent for the game. The algorithm is shown in Algorithm 1 below.

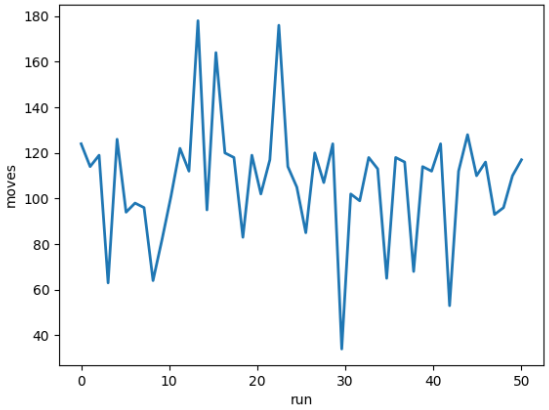
In the algorithm, the advantage is computed by:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \quad (2)$$

Where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$. Similar to the other models above, we also have a replay buffer to shuffle the observations in order to remove the correlations. There are two networks: one is the actor network and the other is the critic network. Actor network is used to choose which direction to move. We used two fully connect layers with ReLU as activation function and end with a softmax function to pick the action that gives the best result. And the critic network has the similar network configuration as the actor network but at the layer, we use another fully connected layer to output a single value to represent the value of the state. We followed the algorithm described (Schulman et al., 2017):



(a) MCTS in train



(b) MCTS in test

Figure 5. Steps played in a game using Monte Carlo Tree Search with width 20 and depth 35

4. Evaluation

Before testing the performance of any model, the first thing we need to have a sense of the environment. As shown in Figure 4 an agent with random policy in the environment can play 29.992 turns in average with an standard deviation of 0.399 steps. And based on our observation, we also found that the tile with value of 256 normally appears in steps between 130 and 260.

4.1. Performance of MCTS

We trained Monte Carlo Tree Search with different depth and width configurations. In Figure 5 we show the Monte Carlo Tree Search with a width of 20 and a depth of 35. The average steps the agent can play with MCTS model is 106 steps in train with a standard deviation of 1.40. While in during the test, the mean steps surprisingley increased to 111 steps per game. The performance seem to grow with search depth and search width. Due to the long training period we did not try to find the optimal parameters, but when search depth is set to 15 and width is set to 8, the average steps is arround 80, which is still a lot better than a greedy agent (which has average step of 50) and random agent (with average step of 30).

4.2. Learning curve

The model in our baseline (Rui, 2015) is trained for the standard 2048 game with a size of 4 x 4. We retained the model descried in the repository on a 3 x 3 board with 100000 episodes and each episodes is a game from start to the end of game. From Figure 6, we can see the steps at the

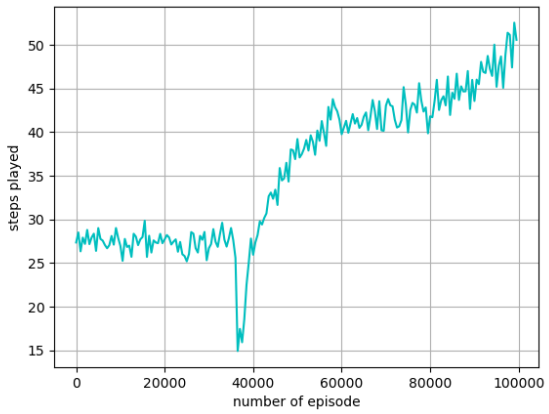


Figure 6. Training curve for DQN of our baseline (Rui, 2015), vertical axis is the number of steps played in a single game and the horizontal axis represents the number of episode

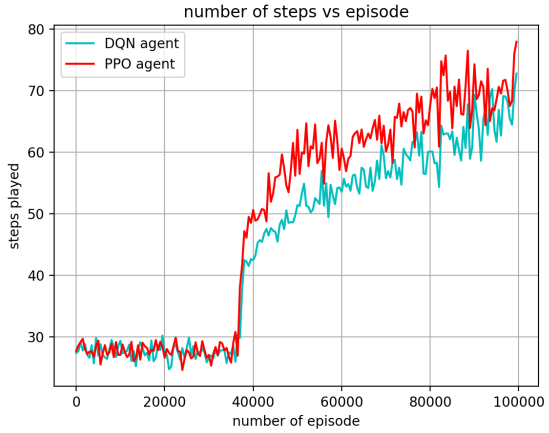


Figure 7. Training curve for DQN and PPO, number of steps played in a single game verse number of episode

beginning is on the par with an random agent but with a low variance. Then the steps played dramatically drop to about 15 steps each game. The reason should be discovered a bit more in the future work but possibly because the divergence of reinforcement learning of nonlinear functions such as neural network (van Hasselt et al., 2018). After the valley, the number of steps start increase with the increase number of game played. This model is able to achieve about 50 steps per game after training 100000 games.

We also trained our own agent implemented with DQN and PPO with the new reward function respectively. From Figure 7 we can see that both model is able to achieve more than 70 steps after 100000 games. The red curve represents the agent with PPO method and the blue curve represents the agent with DQN method. It is clear that, the PPO agent significantly gives a better result than the DQN agent especially after 40 thousand episodes.

As we mentioned in section 3.1.2, the reward function used in (Rui, 2015) is the same as the score difference between two consecutive state. This is inspired by the way how people play the game. In addition, the maximum illegal move the agent can take is a fixed number which is set to 10 moves. Back to the Figure 6, we can see that the valley part, which is around 40 thousand episodes, is about having 15 steps per game. Counting the number of illegal moves, the agent seems to pick one illegal move all the time until the end of game.

With the updated reward function: apply a ReLU like function to dynamically set the threshold of the maximum illegal moves and penalize the illegal move makes the training curve monotonic. To achieve playing 50 steps in a game, our agent can achieve the target within 40 thousand games while our baseline may require more than 100 thousand games to achieve.

4.3. Game performance

Besides the training performance, we are also interested in the test result. We trained the DQN agent for about 4 days with more than 1 million games played, the training curve converged. The agent is able to achieve the tile with 256 with most of the time.

agent	avg score	avg steps	256 probability
our dqn	2627.4	158.7	64.4%
baseline	2783.7	189.2	65.9%

Table 2. Final DQN agent game performance

From table 2, our dqn agent can get a score of 2627.4 in average, 158.7 steps per game with a probability of 64.4% to get the tile of 256. While in comparison, our retrained baseline model (even if it takes more time to converge) also gives a good result. It have a higher value in all these three columns. We can see that the average steps two agent played have a larger different than the other two dimensions. With a little difference in score and the probability to achieve a tile of 256, our agent needs much less steps.

This is due to the fact that we added some human experience to the environment as discussed in section 3.1.2. We encourage the agent to put the tile with largest value at the corners. To some degree, we are telling the agent to merge the tiles with larger values as soon as possible in the sacrifice of losing the game soon.

5. Conclusion and Future work

Through this project, we trained three different agents with Monte Carlo Tree Search Algorithm, Deep Q-learning and Proximal Policy Optimization with our modified game environment. All of these agents are able to achieve a tile of 256 with a high probability. We also compared our deep reinforcement learning models with our baseline mode (Rui, 2015). We find that the updated environment, with a different reward function, our agent converges faster than the original setting. However, in trade of a faster learning, or converging rate, our agent usually gives a lower score than the agent trained with a unmodified reward function. This is due to we added some human experience towards the reward function, which is may not be an optimal solution among the countless situations.

Inspired by the paper (Silver, 2017), we may combine our MCTS model to assist the deep reinforcement networks such as the DQN model. The second thing we need to do is to train the agent in the standard board. Because of the observation space increases by several orders of magnitudes, we need to implement an efficient search algorithm. It is also not possible anymore to attain a significant percentage of all state values through MCTS, which proposes significant difficulty. However, we could start by using some heuristics to initiate scores for un-visited states on which we conclude our MCTS search. This method has been proved to be very effective in some 2048 AIs that make no use of deep learning. Another thing we should consider to prune the MCTS search tree is child node expansion method which is mentioned in (Silver, 2017). We expect the proper use of child node expansion to not only decrease the width of the tree but also generate a better estimate of the state value. Recall that we define the state value in our experiment to be the expected remaining game length, whereas the real state value definition should be the expected remaining game length, given that optimal strategy is taken. Child node expansion gives an estimation of the latter, which is better.

6. Acknowledgements

We thank Professor Chong Li, TA Sam Fieldman and Dr. Chonggang Wang for their expertise and assistance throughout all aspects of our study and for their help in writing the report.

References

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. doi: 10.1109/TCIAIG.2012.2186810.

Coulom, R. Efficient selectivity and backup operators in monte-carlo tree search. In van den Herik, H. J., Ciancarini, P., and Donkers, H. H. L. M. J. (eds.), *Computers and Games*, pp. 72–83, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-75538-8.

Knox, W. B. and Stone, P. Reinforcement learning from simultaneous human and mdp reward. 1:475–482, 2012.

Matignon, L., Laurent, G. J., and Le Fort-Piat, N. Reward function and initial values: Better choices for accelerated goal-directed reinforcement learning. In Kollias, S. D., Stafylopatis, A., Duch, W., and Oja, E. (eds.), *Artificial Neural Networks – ICANN 2006*, pp. 840–849, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-38627-8.

mmcenta. Using deep reinforcement learning to tackle the game 2048. https://github.com/mmcenta/left-shift, 2021.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. 2013. URL http://arxiv.org/abs/1312.5602. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

Mnih, V., Kavukcuoglu, K., and Silver, D. Human-level control through deep reinforcement learning. 518(529–533), 2015.

Rui, Y. 2048 environment and dqn algorithm implementation. https://github.com/YangRui2015/2048_env, 2015.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Silver, D. Mastering the game of go with deep neural networks and tree search. 529(484–489), 2016.

Silver, D. Mastering the game of go without human knowledge. 550(354–359), 2017.

Silver, D. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. 2018.

Tesauro, G. Temporal difference learning and td-gammon. 38(3), 1995.

van Hasselt, H., Doron, Y., Strub, F., Hessel, M., Sonnerat, N., and Modayil, J. Deep reinforcement learning and the deadly triad. *CoRR*, abs/1812.02648, 2018. URL http://arxiv.org/abs/1812.02648.