ISSN: 1803-3814 (Printed), 2571-3701 (Online)
https://doi.org/10.13164/mendel.2022.1.001

MENDEL Soft Computing Journal

Deep Learning and the Game of Checkers

Jan Popič✉, Borko Bošković, Janez Brest

Institute of Computer Science, Faculty of Electrical Engineering and Computer Science, University of Maribor, Maribor, Slovenia

jan.popic1@um.si✉, borko.boskovic@um.si, janez.brest@um.si

Abstract

In this paper we present an approach which given only a set of rules is able to learn to play the game of Checkers. We utilize neural networks and reinforced learning combined with Monte Carlo Tree Search and alpha-beta pruning. Any human influence or knowledge is removed by generating needed data, for training neural network, using self-play. After a certain number of finished games, we initialize the training and transfer better neural network version to next iteration. We compare different obtained versions of neural networks and their progress in playing the game of Checkers. Every new version of neural network represented a better player.

Keywords: *Artificial Intelligence, Deep Learning, Convolutional Neural Network, Reinforcement Learning, Checkers.*

Received: 15 October 2021
Accepted: 27 November 2021
Published: 21 December 2021

1 Introduction

For the past few decades researchers from all around the world were trying to design algorithms with which computer programs could play against and potentially even beat human opponents in certain board games. One such board game that received a lot of attention at the start was checkers. It has 5×10^{20} possible figure positions which, although pale in comparison to chess with approximately 10^{43} possible figure positions, was a high enough number that the game was not simply solvable and yet small enough for researchers to do their research on.

Arthur Lee Samuel was one of the pioneers working in this field. In 1956, he presented his program, which was one of the first programs to successfully play checkers when given only the rules and some basic domain knowledge [4]. It utilized basic min-max tree with certain improvements, which could be perceived as roots of alpha-beta pruning. The mentioned program was later the first program to beat human opponent in 1963.

Idea of basic neural networks, albeit far from the way we know them today, has been around since the beginning of 20^{th} century or longer, but it was still lacking a proper learning mechanism. This was focus of researches for the next few decades [6]. In the last decade, so called deep neural networks have seen a tremendous increase of applications on various different problems across multiple disciplines, which can be greatly contributed to enormous increase of processing power we have available today. Today neural networks are vastly used in various fields, for example biology [1] and medicine [10].

Deepminds programs AlphaGo [7], AlphaGo Zero [9] and Alpha Zero [8] are one such example of using a deep convolutional neural networks as part of a bigger algorithm that can learn to play a game of Go (as well as chess and shogi with Alpha Zero). In our work we will try to replicate the ideas used in mentioned programs and apply them to the game of checkers. Our goal will not be to create unbeatable checkers playing program but only to develop an algorithm which is capable of learning the game of checkers without human interaction and knowledge.

In Section 2, we describe work that is historically important to the game of checkers and further explain algorithms AlphaGo, AlphaGo Zero and Alpha Zero. In Sections 3 and 4, we describe the problem and present our implemented approach. The following Section 5 provides description of our experiments and continues into discussion of our results. Paper is ended with brief conclusion in Section 6.

2 Background and Related Work

In 2007, researchers have declared checkers as a weakly solved game [5]. Their program called Chinook, tgat was being worked on between 1989 and 2007, provided computational proof that from basic starting layout end result of a perfect game is a draw. They achieved this with alpha-beta tree search and extensive endgame database, which contains all possible game variations when there are 10 or less figures on the board. In cases where there are more figures on the board, the program does not play a perfect play.

2.1 AlphaGo

Even though AlphaGo [7] is not directly connected with the game of checkers, it had a huge impact on the world of game artificial intelligence. It was a first

program to beat a professional Go player in 2016, which is something that was unprecedented at the time. AlphaGo utilized Monte Carlo tree search and two independent types of neural networks (NNs) with combination of supervised and reinforced learning. First set of networks, each called a policy network, was used to determine move probability from current board state. Neural networks, p_σ and p_π, with latter being a faster but less accurate version, were trained on a database of 30 million game positions gathered from plays of human players. Third NN, f_ρ, has been initialized to same weights as p_σ had after learning, but it was further improved using reinforced learning on self-play. Second NN type, called value network, is used to predict which player is going to win the game and it was trained using self-play database.

2.2 AlphaGo Zero

AlphaGo Zero [9] upgraded its predecessor even further. The need for human knowledge was removed from the algorithm by removing the database of human plays and utilizing only reinforced learning. Authors have replaced independent neural networks with only one neural network which outputs both move probabilities and value determining the probability for current player wining from current position. Input to the neural network consists of 17 19x19 images stacked vertically, with first 8 images representing current player stones in last 8 moves, next 8 images opposing player stones and last image representing what color is on the move. Authors managed to prove that even approach that does not utilize any human knowledge is able to beat previous state-of-the-art approach from AlphaGo.

2.3 Alpha Zero

Even though AlphaGo Zero achieved stat-of-the-art results regarding the game of Go, it was not easily transferable to other problems. One of the biggest limitations of AlphaGo Zero was the fact that there is no draw in Go, game is either won or lost, but this is usually not the case with other games. It also utilized data augmentation techniques which were only possible because rules of Go are invariant to rotation and reflection. In Alpha Zero [8], authors have removed handcrafted domain specific knowledge affiliated with Go from their algorithm. They applied this new state-of-the-art algorithm to the game of chess and shogi alongside Go. They managed to prove that same algorithm without major alterations can be applied to games of Go, shogi and chess and surpass previous state-of-the-art algorithms in each field.

3 Problem Formulation (Equations and Variables)

Our work is based on the ideas of AlphaGo [7], AlphaGo Zero [9] and Alpha Zero [8] which we encourage the reader to study for more details. We used a part of existing framework [2] which already contained certain things needed for our work.

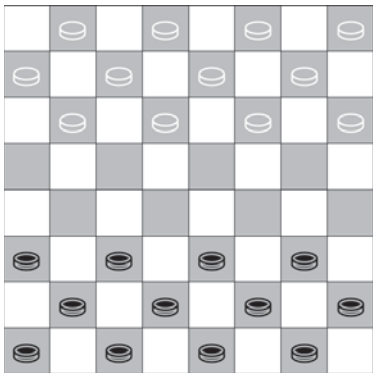


Figure 1: Starting state of the board in German checkers.

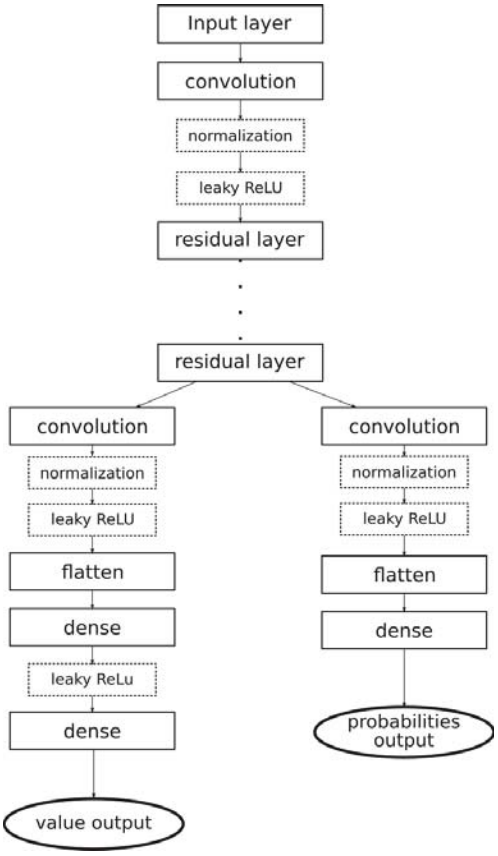


Figure 2: Neural network model.

We decided to focus on rules of German checkers in which the game is played on 8x8 board with interchanging black and white squares. There are 12 white and 12 black playing figures that are coin like shaped. Each player has their figures arranged on black squares in first 3 rows closest to them (see Fig.1). Game is always started by the player with black figures and is continued by the opponent. Regular game figures can only move diagonally on black squares towards opponent's

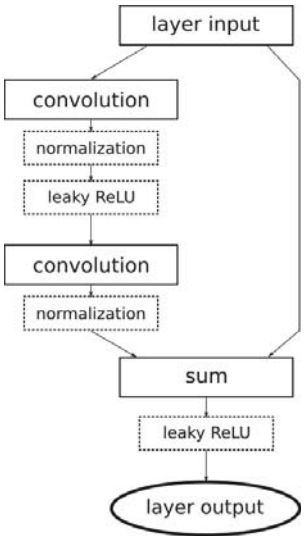


Figure 3: Residual layer model.

side one square at time. If there is an opponent's figure in the path of the move and square behind that figure is vacant, the player must capture the opponent's figure by jumping over it and removing it from the playing board. If multiple such successive captures exist, the player must perform them all in one move.

If the player reaches opponent's side of the board, their (player's) figure is knighted and hence has the ability to move diagonally forward and backward. Game is won if the opponent has no figures left on the board or is out of possible legal moves. To prevent infinite games, we have added a rule that game ends in a draw if no pieces have been captured in last 40 moves – something that is usually declared by the game referee.

Our work can be divided into 3 bigger concepts: player agent, Monte Carlo Tree Search (MCTS) and neural network.

3.1 Player Agent

Each agent consists of its own MCTS and its own neural network. Agent gets a list of possible legal moves it can make on its turn from the game environment. It also has a whole overview of the game state at any given time – meaning it has full knowledge where the opponent has its figures and what type of them they are.

During every turn the agent uses MCTS algorithm to select the best move from the list. We utilize random move selection – in which the MCTS part is skipped – for the first `TURNS_UNTIL_TAU0` game moves to further explore new and different game scenarios. This is only used in the learning phase of the approach and not in the tournament (playing phase).

3.2 Monte Carlo Tree Search

Before the agent selects a move from the current game state s, we first run `MCTS_SIMS` iterations of MCTS algorithm. In our game tree each node, which represents a certain move a, contains a number of node visits $N(s,a)$, its value $Q(s,a)$ and probability of selecting that move $P(s,a)$ which is received from neural network. Current node value $Q(s,a)$ represents mean value of its branch and is calculated from its leaf value $V(s_L^i)$ which is returned by neural network, $l(s,a,i)$ which represents if leaf i was visited from current node and current node visit counter $N(s,a)$ (see Eq.1).

$$Q(s,a) = \frac{1}{N(s,a)} \sum_i l(s,a,i) V(s_L^i) \qquad (1)$$

We start from the root and continuously select moves a that maximize our score $S(s,a) = Q(s,a) + u(s,a)$ where $u(s,a) \propto \frac{P(s,a)}{1+N(s,a)}$ and is used to improve exploration of moves ,which have not been selected often or moves that already have a high enough prior probability $P(s,a)$ of being selected. During each node visit, we update its values and evaluate the position with neural network if we reached the leaf. After `MCTS_SIMS` iterations of tree traversal, we select a move (first childs of root node) which has the highest value $Q(s,a)$.

3.3 Neural Network

Structure of our neural network follows the one described in [8]. The input in our neural network is a matrix of size 8x8 which describes current game state. After the input layer, there is a convolutional layer with 75 kernels of size 4x4, followed by 5 residual layers. Overall structure is represented in Fig. 2.

Each residual layer consists of 2 convolutions with 75 kernels of size 4x4 followed by addition of this convolutions and layer input. Residual layer is represented in Fig. 3. After every convolutional layer we utilized batch normalization to speed up the learning and mitigate overfitting [3].

Our neural network has 2 outputs. First is the result of convolution layer, flattening layer and two dense layers which reduce the dimension to one value. This value represents value V of current board state. Second output is achieved by convolution layer, that is again followed by flattening and dense layer, which reduces the dimension to vector of size 64. This represents probability for each position.

Neural network utilizes leaky ReLU activation function and stochastic gradient descend with momentum to adjust network weights and kernels.

4 Learning Through Self-Play

Our work consists of learning through self-play between two versions of neural network. General flow chart can be seen in Fig. 4. We generate data needed for learning by playing a number of games, provided by parameter `EPISODES`, between agent using current neural network

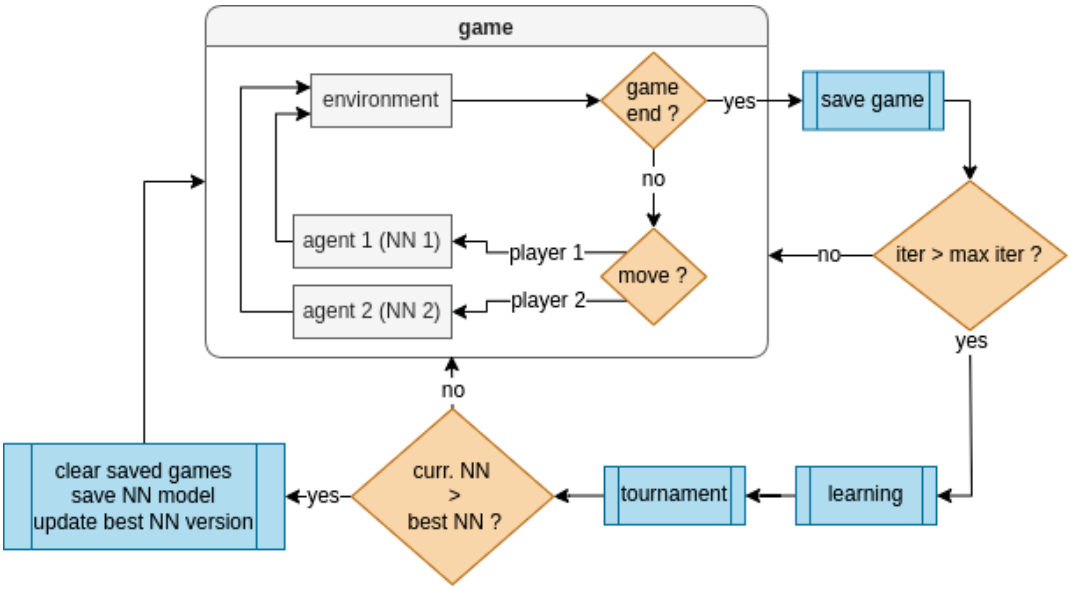


Figure 4: General flow chart.

Table 1: Algorithm parameters.

Parameter	Value	Description
EPISODES	50	number of self-play games for data creation
MCTS_SIMS	70	number of MCTS iterations
TURNS_UNTIL_TAU0	25	number of moves after which the game is played deterministic
BATCH_SIZE	256	batch size used for learning
LEARNING_RATE	0.1	learning rate
MOMENTUM	0.9	learning momentum
EVAL_EPISODES	20	number of games played in validation phase
SCORING_THRESHOLD	1.3	scoring factor in validation phase

and agent using best neural network (both neural networks are the same in the first iteration). During this data creation phase, every game is started by a random agent thus eliminating any advantage starting player would have. After each game, we save game states, game winner and move probabilities from MCTS.

After EPISODES games have been finished, we initialize the learning phase. During this phase, we use generated data to train current neural network with reinforcement learning combined with input batches.

When learning phase is successfully completed, we start a tournament between the agent using newly trained neural network and the one using currently best neural network. During tournament EVAL_EPISODES games are played, and each agent is scored according to the chess scoring (3 points for a win, 1 point for a draw and 0 points for a loss). After tournament completion we check if agent using newly trained neural network has obtained more points (ratio defined with SCORING_THRESHOLD) than agent using currently the best neural network. If that is the case, then we set this neural network as currently best, discard the old one and repeat the process.

5 Experiments

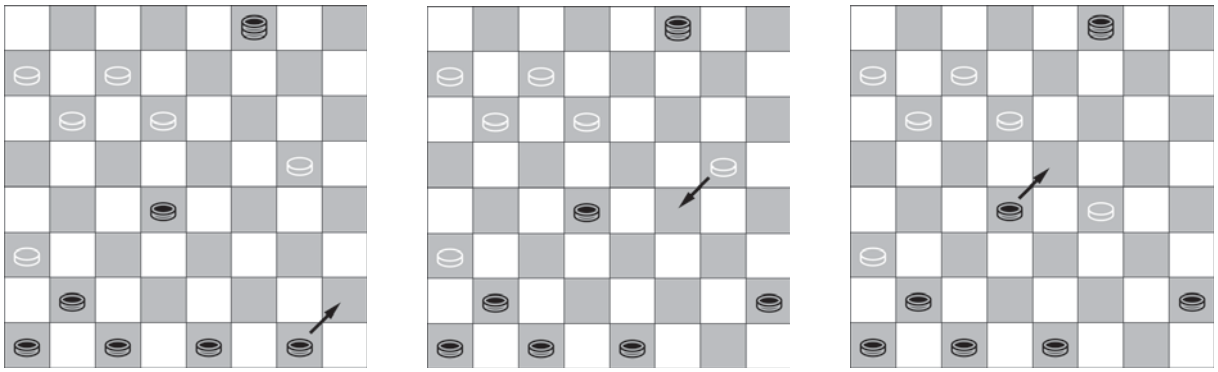
In this section we describe our experiment and results, followed by a brief discussion of certain moves of agents.

We ran our algorithm, as described in previous section, with parameters as seen in Table 1. Instead of discarding NN model when better one was found, we saved that version for later use. After approximately 2 weeks, we obtained 9 different versions of NN model, each being marked as better than the previous one. We ran 50 games between agent using NN model marked as the best (v9) and agents using every other NN version. Every game was scored according to the chess scoring. Number of wins (W), draws (D) and losses (L) of 9^{th} NN version versus other versions as well as final score (S) can be seen in Table 2.

As expected, final scores against lower/worse versions are generally higher than those against higher/better versions. Number of wins generally decreases and number of losses generally increases, again as expected. We can also see that when v9 played against itself (see column v9 in Table 2) it

Table 2: Games of agent using 9^{th} version of neural network against other versions (v0 to v8) of neural networks and itself (v9).

	v0	**v1**	**v2**	**v3**	**v4**	**v5**	**v6**	**v7**	**v8**	**v9**
W	32	27	26	33	29	26	18	11	14	17
D	13	18	19	10	17	13	16	20	14	16
L	5	5	5	7	4	11	16	19	22	17
S	109	99	97	109	104	91	70	53	56	67



(a) 30^{th} move. (b) 31^{st} move. (c) 32^{nd} move.

(d) 33^{rd} move. (e) 34^{th} move. (f) 35^{th} move.

Figure 5: Moves 30 to 35 of the game between 9^{th} version (black) and 3^{rd} version (white) of NN.

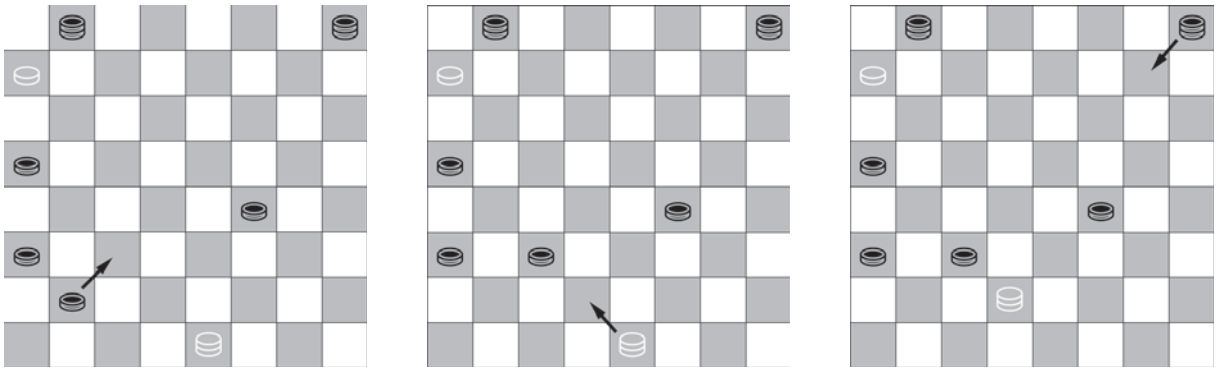
obtained the same number of wins and losses, which is on par with the idea that two equivalent players will have a 50/50 chance of winning/losing.

When observing some games of 9^{th} NN version we could notice certain moves which seemed like carefully calculated plays. Agent properly recognized that knighted figures have bigger impact in the game and therefore defended his and attacked opponent's knighted figures. Figure 6 shows agent's ability to sacrifice his figure to capture opponent's knighted figure. In move 57 (Fig. 6c) black player chose to move irrelevant figure when it could have actually moved his other figure to avoid capture, but it rather chose to sacrifice it and capture opponent's knighted figure in move 59 (Fig. 6e). This also created a situation in which black player wins, no matter which move the white player makes.

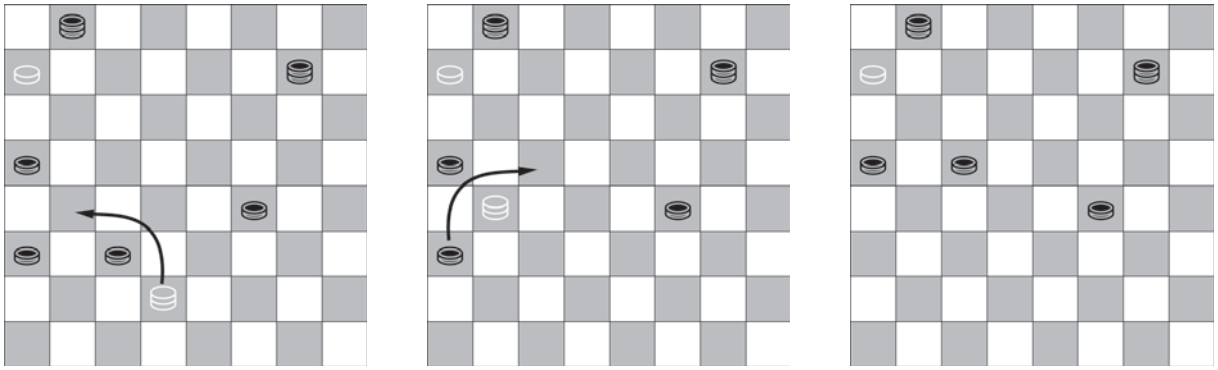
As seen on Fig. 5c, agent also utilized opponents figures to prevent capturing of own figures (space behind the figure must be empty in order to capture it). First version of NN on the other hand did not display such qualities and as it can be seen from Table 2, 9^{th} version won most of the time.

6 Conclusion

With this paper we managed to demonstrate the ability of presented approach, which is based on AlphaGo Zero and Alpha Zero, to learn the game of Checkers when given only s set of game rules. This approach is only given a set of game rules, and then using self-play generates needed data to train newer models of neural network. After 2 weeks of runtime we obtained 9 different versions of neural network, each being better than then previous one. We compared the best version of neural network to other versions in the final comparison and



(a) 55^{th} move. (b) 56^{th} move. (c) 57^{th} move.



(d) 58^{th} move. (e) 59^{th} move. (f) 60^{th} move.

Figure 6: Moves 55 to 60 of the game between 9^{th} version (black) and 3^{rd} version (white) of NN.

noted that the last version is superior than the previous ones. When analyzing the games played, it was obvious that moves are more "intelligent" compared to the initial few versions where moves were without any visible strategy. We can see some discrepancies with regards to the score achieved by the first few versions; this is most likely due to noise and could be addressed in our future work with optimization of the parameters (`EPISODES`, `EVAL_EPISODES` and `SCORING_THRESHOLD`).

Acknowledgement: This work was supported by the Slovenian Research Agency (Computer Systems, Methodologies, and Intelligent Services) under Grant P2-0041.

References

[1] Baek, M., DiMaio, F., Anishchenko, I., Dauparas, J., Ovchinnikov, S., et al. Accurate prediction of protein structures and interactions using a three-track neural network. *Science 373*, 6557 (2021), 871–876.

[2] Foster, D. Deep reinforcement learning. Available at https://github.com/AppliedDataSciencePartners/DeepReinforcementLearning, 2018.

[3] Ioffe, S., and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32Nd International Conference on Machine Learning - Volume 37* (2015), ICML'15, JMLR.org, pp. 448–456.

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

[5] Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., and Sutphen, S. Checkers is solved. *Science 317*, 5844 (2007), 1518–1522.

[6] Schmidhuber, J. Deep learning in neural networks: An overview. *Neural networks 61* (2015), 85–117.

[7] Silver, D., et al. Mastering the game of go with deep neural networks and tree search. *Nature 529* (Jan 2016), 484 EP.

[8] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science 362*, 6419 (2018), 1140–1144.

[9] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., et al. Mastering the game of go without human knowledge. *Nature 550* (Oct 2017), 354 EP.

[10] Srinidhi, C. L., Ciga, O., and Martel, A. L. Deep neural network models for computational histopathology: A survey. *Medical Image Analysis 67* (2021), 101813.