JengaZero: Playing Jenga like a Grandmaster

Pranav Rajbhandari
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213
prajbhan@andrew.cmu.edu

Sam Rosenstrauch
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213
srosenst@andrew.cmu.edu

Abstract

Jenga is a tabletop game that requires a blend of strategy and dexterity to master. While there exist projects handling the manipulation required to play the game, solving the strategic aspect of Jenga goes largely unexplored. In this project, we utilize Deep Learning methods to create an agent to strategically play the game of Jenga. We approximate the game with a static numerical model, allowing us to capture complexities from randomness in block placement without the computational cost of running a physics model. We then create various agents to play the game, including a random agent, an agent using a Deep Q-Network, an agent using a Monte Carlo Tree Search, and an agent inspired by the Chess AlphaZero engine. We train the agents through games in the static numerical model, and evaluate their performance using the Elo ratings. We found that the AlphaZero algorithm outperforms the others, followed closely by MCTS.

1 Introduction

To play Jenga well, players must both strategically decide which block to try removing, and carefully remove and replace the block chosen without causing the tower to collapse. Manipulation of the blocks is certainly a crucial aspect of the game. In fact, a project from MIT created a robot to remove and place Jenga blocks with visual and tactile data [4]. In contrast to this study, we focus on the largely unexplored strategic aspect of the game.

In our project, we focus on utilizing Deep Learning methods for Jenga gameplay. We aim to create an AI that analyzes a representation of a tower to make informed decisions of the optimal moves. The agent will be trained using samples of games that it plays against itself as well as games against other agents. We model the game with a static numerical model, allowing for randomness in block placements while avoiding the computation cost of a physics simulator. We plan to train agents on this version of the game to speedup training.

We evaluate a trained agent's performance through their games against other trained agents, as well as designed agents such as one that randomly chooses an action. We use metrics such as the Elo system [3] and win rate to compare agents.

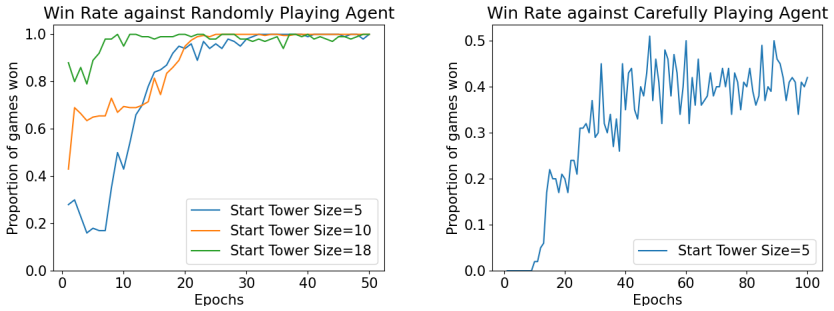
Our implementation was done in Python and is available as a repository [7].

2 Background (Prior Results)

In prior experiments, we implemented the random agent (Section 5.1), the 1-step agent (Section 5.2), and the DQN baseline (Section 5.3). Since we only had three agents, a method as sophisticated as ELO was not necessary, so we inspected the win rates of DQN against the other agents. We found that a converged DQN wins almost always against a random agent (Figure 1(a)), and won around

Not submitted to 37th Conference on Neural Information Processing Systems (NeurIPS 2023). Feel free to distribute.

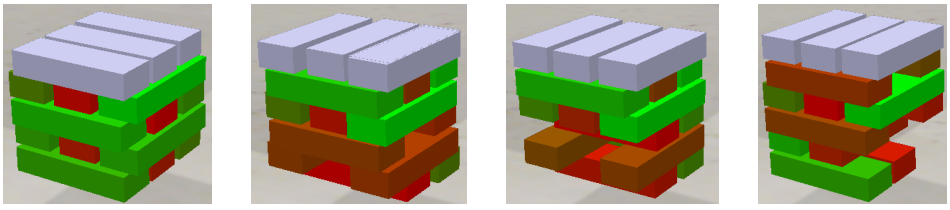
half of the time against the 1-step agent (Figure 1(b)). We also inspected the Q-values the network returned (Figure 2) and verified that while the DQN learned to not immediately knock over the tower, it did not learn a more sophisticated Nim-like strategy [2], and instead tried to make the game last long enough for the opponent to make a mistake and lose.



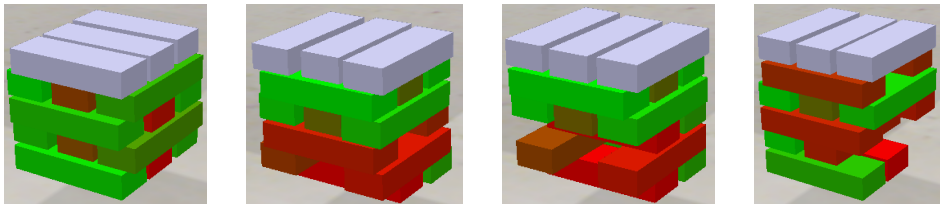
(a) Performance of three DQN agents against Random agent

(b) Performance against 1-step agent

Figure 1: Performance of various DQN agents against Random and 1-step agents during training



(a) DQN against Random agent



(b) DQN against 1-step agent

Figure 2: Q-values of removing blocks from various towers, obtained from a DQN agent. Green blocks have a high Q-value, and Red have low Q-value. Gray blocks are invalid to remove

3 Related Work

The strategic game of Jenga is largely unexplored, with the only machine learning approach solving the manipulation problem [4]. However, generic game playing techniques can be implemented for a solution to Jenga strategy.

3.1 Deep Q-Network

Q-learning is a model-free reinforcement learning algorithm that learns the value of an action in a particular state [10]. It uses a Q-table, where each row represents a state and each column represents an action. The Q-values are updated using the following formula:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \mathbb{E}_{s'}[\max_{a'} Q(s',a')|s,a] - Q(s,a))$$

where (s,a) are the current state and action; (s',a') are the next state and action; r is the reward; α is the learning rate; and γ is the discount factor.

In Deep Q-Networks (DQNs), a deep neural network approximates the Q-value function, denoted as $Q(s, a; \theta)$, where θ represents the network's parameters. The Q-function represents the quality (expected reward) of taking action a in state s, given the agent's policy is to choose the action with the maximum Q-value. The network is trained to minimize a series of loss-functions that change at each timestep [8].

$$L_i(\theta_i) = \mathbb{E}_{s,a}[(r + \gamma \mathbb{E}_{s'}[\max_{a'} Q(s', a'; \theta_{i-1})|s, a] - Q(s, a; \theta_i))^2]$$

Typically, a DQN is trained through examples sampled from a **rollout**, a trajectory taken through the states. The rollout can either be obtained from the agent itself (adding some stochastic noise to exploration), or viewing other agents in the environment. In our experiments, we obtain these rollouts through both self-play and play against another agent. An epoch consists of collecting training data through rollouts, then optimizing parameters using the Q-value update.

3.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) consists of four key phases that are repeated as part of its execution cycle:

1. **Selection:** Starting from the root node, the tree is traversed to a leaf node using a policy that balances exploration of less-visited nodes and exploitation of known good paths.
2. **Expansion:** Unless the selected leaf node ends the game (a win, loss, or draw), one or more child nodes are added to expand the search tree. Each child node represents a possible future move in the game or decision in the domain.
3. **Simulation:** A simulation is run from the state represented by the new node to a terminal state using a default (usually random) policy. This simulation, also known as a play-out, helps to estimate the potential of the node without fully exploring each path.
4. **Backpropagation:** The results of the simulation are propagated back up the tree, updating the statistics (such as win/loss ratios) of the nodes along the path traversed during the selection phase.

The search tree in MCTS is composed of nodes representing game states or decision points, and edges representing the moves or actions that transition from one state to another. MCTS can be used with an upper-confidence bound. Starting from a parent node, the agent will choose the child node i that maximizes $v_i + C\sqrt{\frac{\ln N}{n_i}}$, where N is the number of times the parent node is chosen, v_i is the empirical average value of child i, and n_i is the number of times child i is chosen from the parent. The constant $C > 0$ is called the exploration constant, and it can be made larger to encourage more exploration [11].

3.3 Alpha-Zero

Alpha-Zero, developed by DeepMind Technologies, represents a significant leap forward in the field of AI. Unlike its predecessors, Alpha-Zero learned to play deterministic games without any human data or guidance, relying solely on reinforcement learning and self-play [9].

Its algorithm is an extension of MCTS, with edits to the Selection and Simulation stages.

Given a state embedding for the game, a value network and a policy network are jointly trained. The value network tries to output the expected value of a state, and the policy network tries to output the optimal policy given a state. MCTS utilizes the value network in place of the Simulation step, directly taking the value instead of making a random play-out. MCTS utilizes the policy network in the Selection step, pushing the exploration policy towards the output of the policy in order to explore more 'useful' parts of the game tree. From a parent node, the action a chosen for state s maximizes $v_i + C \cdot P(a|s) \cdot \frac{\sqrt{N}}{1+n_i}$, where v_i, C, N, and n_i are defined the same as in MCTS (Section 3.2).

During updates, a training example is an explored MCTS tree. The root node has a value (defined as the max of the values of each possible action), and the value network is updated towards this value. The root node also has a policy distribution, calculated from the Q-values through a softmax. The policy network is updated towards this distribution.

The motivation behind Alpha-Zero comes from the fact that the MCTS algorithm outperforms the policy that it takes in its selection phase. This is since the method explores the search tree guided by the policy, so it always performs at least as well as the exploration policy. Thus, if the exploration policy is continually updated towards the policy derived from MCTS, the policy should gradually improve.

4 Methods (Gameplay)

4.1 Representation

Our goal was to model a real Jenga tower in a way that is interesting to play (i.e. the block positions have some randomness) while also being computationally easy to evaluate for stability. The method we chose was to represent the positions and orientations of each block, and numerically estimate the stability of the tower (whether the tower would fall or not). We chose this method as it allowed for randomness in block placement while still being relatively easy to compute.

4.1.1 Tower

We represent a Jenga tower (i.e. state of the game) as a set of positions and orientations of blocks. Each block is exactly the size of a standard Jenga block ($.015 \times .025 \times .075$ m). We assume that each block has uniform density, and that each block lies flat (i.e. the rotation is only around the z axis).

We add the restriction that the z values must be in $\{.075/2 + .075n : n \in \mathbb{N}\}$ (representing the fact that each block is an exact multiple of $.075$ above the lowest block). Additionally, there are at most 3 blocks at every possible z value.

4.1.2 Layers/Subtowers

We partition the tower into **layers** in a natural way, through an equivalence relation of blocks with the same z-coordinate. We denote layer 0 (denoted L_0) as the set of blocks touching the ground (at $z = .075/2$), and the layer L_{i+1} as the set of blocks exactly $.075$ above L_i. From our assumptions, this captures all blocks.

For a tower with layers $L_0, L_1, \ldots, L_N$, the **subtower** from layer i is the tower with layers $L_i, L_{i+1}, \ldots, L_N$ in that order. We decrease the z coordinate of each block by $i * .075$ so that the first layer is on the ground.

4.1.3 Terminal State (Fallen Tower)

We say that the Jenga game with tower $T = L_0, \ldots, L_N$ reached a terminal state (i.e. is **unstable**) if for some i, the x, y coordinates of the center of mass of the subtower from layer $i + 1$ lies outside of the convex hull of the x, y projection of layer i.

4.2 Gameplay

Since many variations of the Jenga rules exist, we choose to use the following version [6]:

A player's move consists of removing one block from any layer of the tower (except the one below an incomplete top level), then placing it on the topmost layer in order to complete it. We represent this in our static numerical model by considering the tower with one block removed, and the resulting tower with a block added. We add a small amount of Gaussian noise when adding the block represent the randomness of real Jenga. If either of the towers are unstable, we assume the tower fell on this turn, and the game ends with the current player losing.

We do not allow an agent to remove the last block on any level, and if an agent has no possible moves at the beginning of their turn, this counts as a loss (similar to the mechanics of Nim).

To make the game more realistic, each layer is assigned a probability of collapsing, determined by the directed distance between the projected center of mass of the subtower above that layer and the convex hull of the layer itself. We treat the likelihood of each layer falling as independent random events based on these probabilities. During gameplay, after a block is removed and then placed, we randomly assess whether this action causes the tower to fall.

4.3 Tower Representation

4.3.1 Basic Embedding

We hand picked features of the tower, and used these in a feed-forward network. Our tower embedding included the center of mass of each subtower, the height of the tower, the binary encoding of the presence of a block at each position, and the count of each 'layer type'.

4.3.2 Nim Embedding

Since Jenga is similar to a game of Nim [2], a game whose state can be summarized with features in $\mathbb{Z}/k$, one embedding choice would be to include 'counts' of various tower features modulo k for various k. The features we included were the height of the tower, the number of blocks on the top layer, and counts of each layer type (where a layer type is the presence of each block up to reflection) in the tower. We use $k \in \{2, 3\}$, as we do not expect the dependence of a Nim game with two players and 'piles' of size 3 to be outside these values.

4.3.3 State-Action Embedding

Most Deep Learning game-playing algorithms use their networks to predict the Q-value of state-action pairs. We will do the same, but an important modeling choice is how to represent a Jenga state-action pair as a real vector input. We embed the state of the tower after removing a block using the Basic Embedding, as well as the state of the tower after placing the block. We also add a binary representation of the action. The concatenation of all these vectors is our state-action embedding.

4.4 Hindsight Experience Replay

Deep Reinforcement Learning tasks can be challenging, as they can suffer from lack of learning due to sparse rewards. Using a **Replay Buffer** is a way to alleviate this, where while collecting environment transitions to train on, we keep the most recent examples in a replay buffer. When the optimization calls for training steps, we randomly sample from the buffer to create a batch. This improves and stabilizes convergence by allowing us to revisit important training examples many times [1].

4.5 Metrics

Bots are given Elo ratings, similar to chess [3]. The Elo system stochastically estimates the skill level of an agent. Suppose a score of 1 represents a win and 0 represents a loss, and consider two agents A and B. The expected score of A against B is predicted to be $E_A = \sigma(\frac{R_A - R_B}{C})$ for some constant C. After a match, the rating R_A is updates as follows:

$$R'_A = R_A + K \cdot (S_A - E_A)$$

where S_A is the score and K is some learning constant. Matches can be done in batch as well and the score will be defined as the sample win-rate. For this study, we used $K = 69$ and $C = 420$, with Elos initialized to 1000.

5 Methods (Agents)

We created various learning agents and non-learning agents to compare their relative performances.

5.1 Random

This agent picks a random valid move.

5.2 1-Step

This agent picks a random safe move. It imagines removing and perfectly placing a block, and checks whether the towers are stable. If there are no stable moves, it picks a random move.

5.3 DQN (baseline)

Our DQN was implemented as a simple 1-layer feed-forward neural network with RELU activation. We had a hidden layer of 256 units, and used smoothed L1 loss, as some studies indicate this helps prevent gradient scaling problems [5]. Our input was the state-action embedding (Section 4.3.3) using the basic tower embedding (Section 4.3.1). We train the DQN using a Replay Buffer, and get training examples through both games the agent plays against itself, and games against the Random agent.

The networks were trained with a batch size of 128 and a buffer size of 1024. They were trained with 50 epochs where an epoch consists of a sample game and one batch update. The games were either self play or played against the Random opponent.

5.4 Monte Carlo Tree Search

We implemented a version of MCTS (Section 3.2) with an exploration constant of $2\sqrt{2}$, as the reward ranged between -1 and $+1$. Since the environment's rewards are randomized (Section 4.2), a direct implementation of MCTS would not be a good model of the gameplay. Thus, we slightly modified the Selection phase to recalculate the probability of a tower falling over each time it is passed. We also modified the Backpropogation stage to negate the rewards as they are passed up the tree, since in an adversarial game, the value of the resulting tower is observed by the opponent.

The MCTS policy is determined by the highest valued action after 1000 iterations of the algorithm.

5.5 Jenga-Zero

Our Jenga-Zero models are inspired by the Alpha-Zero algorithm developed by DeepMind Technologies, representing a significant advancement in AI. Jenga-Zero utilizes two neural networks: a policy network and a value network, both incorporating embedding techniques for training.

To address the reward range from -1 to $+1$, we scale these inputs by 2000. This scaling helps the policy network to prioritize a uniform distribution among top-ranking candidates with closely similar values.

In our implementation, the value network and the policy network shared parameters, and consisted of a feed-forward network with two hidden layers of 128 units. We assigned one dimension of output to the value calculation, and the rest of the dimensions to the policy. The policy output activation was softmax, and was optimized through cross-entropy loss. The value had no activation, and was optimized using squared-error loss.

The networks were trained with a batch size of 128 and a buffer size of 1024. They were trained with 420 epochs where an epoch consists of a sample game and one batch update.

The JengaZero policy is determined by the highest valued action after 1000 iterations of the algorithm.

5.5.1 Jenga-Zero Nim

This agent is trained via an alpha-zero algorithm, with the Nim embedding (Section 4.3.2) as input.

5.5.2 Jenga-Zero Union

This agent is trained using the alpha-zero algorithm and a 'union embedding' (concatenation of Sections 4.3.1 and 4.3.2) as input.

6 Results

Playing each pair 100 times, allowing each agent to play first half the time yielded the following win rates and Elo ratings

Random	1-Step	DQN-Nim	DQN-Basic	MCTS	JengaZero-Union	JengaZero-Nim
567	1160	562	595	1282	1459	1374

Table 1: Elos of agents after 100 games

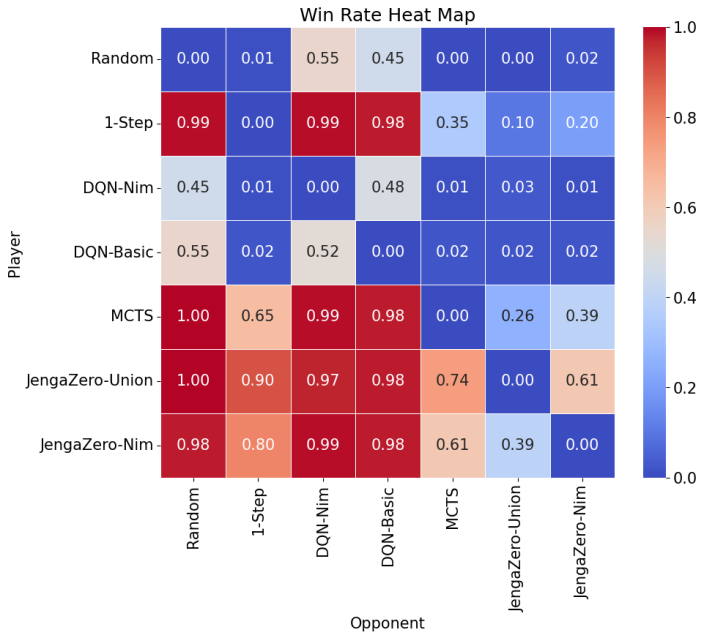


Figure 3: Win rate heat map

7 Discussion and Analysis

7.1 Algorithms

The results show that the JengaZero algorithm outperforms all other algorithms, followed closely by MCTS. The DQN algorithm performs poorly, having an Elo similar to the Random agent (Table 1).

7.1.1 JengaZero

We can see from the win rates (Figure 6) that the JengaZero agents won consistently against DQN and Random opponents, won relatively often against the 1-Step opponent, and won about 60% to 70% of the time against MCTS.

The most interesting case to inspect is JengaZero and MCTS, as JengaZero is an extension of the MCTS algorithm. We can see that the learned value and policy networks did yield an effective update to MCTS, as the difference in performance can only be explained by the networks causing the algorithm to explore more useful areas of the search tree. This result is consistent with AlphaZero's performance on deterministic games, indicating that the extension into non-deterministic games still yielded a useful policy.

7.1.2 DQN

We noticed that the DQN performed very poorly. We theorize this is a result of a lack of sufficient training examples for end-game states, as most of its replay buffer consisted of short games against itself and the random agent. A potential improvement of our methodology could be to select a training set of longer games, or simply to train for longer using self-play. Another issue could be that the state encoding we used for DQN was either too high dimensional or too difficult to extract useful information from. This would be fixed by a more useful state space (explored in Section 8.1).

7.2 State Embeddings

We also noticed that for JengaZero, the Nim-embedding performed competitively to the Union-embedding, although it was given a strict subset of the information. This supports our hypothesis that the game of Jenga can be reasonably approximated as a game of Nim, since an agent trained on data in $\mathbb{Z}/k$ performed similarly to an agent trained on more data.

8 Limitations/Future Work

8.1 State Encodings

The encoding of a tower into a vector is vital to how deep networks transform the state to create a policy. In our work, we handpicked features and used those to embed each tower. However, future work could embed the tower in a more complex way to improve the results of all the methods, and to better capture the complexities in the block placements.

8.1.1 Sequence Model

Viewing a tower as a sequence of layers suggests that a sequence model could be used to embed the tower as a real vector. Allowing the gradient of each training method to flow through the embedding would create a full end-to-end algorithm that learns the best move from the raw tower data. Possible sequence techniques include recurrent neural networks or transformer models.

8.1.2 Convolutional Neural Net

Viewing a tower as a 3-D density field suggests that a convolutional neural network could be used to embed the tower as a real vector. The input would be a 3-D boolean matrix indicating the presence of a block at each spatial coordinate. Applying a 3-D CNN to this matrix would effectively embed this into a vector, and gradients from the policy training would be able to flow through the CNN layers to train the embedding as well.

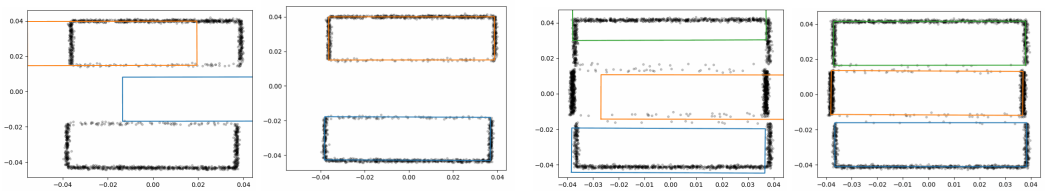
8.2 Network Architecture

Our experiments used a single network architecture for each model, not optimizing the size and structure of the networks used. A potential future direction would be to fine tune the network architecture for each model, and perform a similar experiment comparing the performances of the fine-tuned models.

8.3 Sensing Data

In our work we assumed that we would know the position of each block in the tower. In real applications, this information would have to be inferred through sensors. The method of predicting the block locations from sensing data is a potential extension of our work.

8.3.1 Preliminary Results



(a) Pointcloud of two blocks (b) Pointcloud of three blocks

Figure 4: Initial and final solutions of block positions given pointcloud data

We slightly explored this, and created an effective way to estimate the block locations from a 2-D pointcloud. We parameterized each block (x_i, y_i, θ_i) including location and orientation. We then obtained the outlines of the blocks differentiably, and defined a loss function as the mean L1 loss of each point to the outline of the nearest predicted block. We then performed gradient descent on this. For each pointcloud we tried gradient descent using 1, 2, and 3 blocks, and predicted the number of blocks based on the resulting loss (we noticed that the loss would sharply decrease until the correct number of blocks).

References

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight Experience Replay, 2018.

[2] Charles L. Bouton. Nim, A Game with a Complete Mathematical Theory. *Annals of Mathematics*, 3(1/4):35–39, 1901.

[3] Arpad E. Elo. *The Rating of Chessplayers, Past and Present*. Arco Pub., New York, 1978.

[4] N. Fazeli, M. Oller, J. Wu, Z. Wu, J. B. Tenenbaum, and A. Rodriguez. See, feel, act: Hierarchical learning for complex manipulation skills with multisensory fusion. *Science Robotics*, 4(26):eaav3123, 2019.

[5] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015.

[6] Jenga. Jenga Rules. `https://www.jenga.com/about.php`.

[7] Pranav Rajbhandari and Sam Rosenstrauch. JengaZero. `https://github.com/pranavraj575/jengazero`, 2024.

[8] David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *DeepMind*, 2013.

[9] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, page 354–359, Oct 2017.

[10] Christopher Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[11] Mark H.M. Winands. Monte-Carlo Tree Search.