Learning to Search in Reinforcement Learning

Ioannis Antonoglou

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

CoMPLEX

University College London

March 7, 2023

I, Ioannis Antonoglou, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

In this thesis, we investigate the use of search based algorithms with deep neural networks to tackle a wide range of problems ranging from board games to video games and beyond. Drawing inspiration from *AlphaGo*, the first computer program to achieve superhuman performance in the game of Go, we developed a new algorithm *AlphaZero*. *AlphaZero* is a general reinforcement learning algorithm that combines deep neural networks with a Monte Carlo Tree search for planning and learning. Starting completely from scratch, without any prior human knowledge beyond the basic rules of the game, *AlphaZero* managed to achieve superhuman performance in Go, chess and shogi. Subsequently, building upon the success of *AlphaZero*, we investigated ways to extend our methods to problems in which the rules are not known or cannot be hand-coded. This line of work led to the development of *MuZero*, a model-based reinforcement learning agent that builds a *deterministic* internal model of the world and uses it to construct plans in its imagination. We applied our method to Go, chess, shogi and the classic Atari suite of video-games, achieving superhuman performance. *MuZero* is the first RL algorithm to master a variety of both canonical challenges for high performance planning and visually complex problems using the same principles. Finally, we describe *Stochastic MuZero*, a general agent that extends the applicability of *MuZero* to highly stochastic environments. We show that our method achieves superhuman performance in stochastic domains such as backgammon and the classic game of *2048* while matching the performance of *MuZero* in deterministic ones like Go.

Impact Statement

Artificial intelligence (AI) studies the problem of constructing intelligent agents that can reason and make optimal decisions in an unbounded range of tasks. Such systems can have an unprecedented impact on our societies and bring immense benefits to humanity. They can aid us at tackling some of the toughest challenges we face, from global warming and economic inequality to the widespread availability of healthcare and education. They can also serve a more esoteric human need, that of understanding the inner workings of our own minds. This thesis attempts to contribute towards the advancement of the field by proposing three novel general algorithms which have achieved unrivaled superhuman performance in a series of complex tasks that have been heavily used in the field of AI to measure the capabilities of intelligent systems. The first algorithm, *AlphaZero*, conclusively demonstrated the power of deep learning and tree search in computer board game research and has already had an immense impact on the field. *MuZero* has illustrated how AI systems can build internal world models, purely by interacting with their environment, and effectively employ them for planning in exceedingly complex domains. This ability has been long considered as essential for the development of intelligent systems, but had failed to materialize in practice. *Stochastic MuZero* showed how such systems can remain resilient in the face of high levels of stochasticity present in the environment dynamics.

At the same time, the generality of these approaches makes them applicable to a wide range of scientific and commercial problems. Various scientific communities in chemistry [105], quantum computing [30] and optimization [73] have already adopted *AlphaZero* to accelerate progress in their respective fields. *MuZero* has

already been incorporated into Tesla's self-driving system [126], a real world AI application with tremendous potential for the future of transportation. Moreover, it has been applied to the problem of video compression at an internet scale within YouTube's custom compression algorithm [80].

This line of work has opened up new research avenues for the development of better algorithms. In the computer board games research, the current best Go and chess engines are based on the AlphaZero algorithm [91]. It has sparked a renewed interest in model-based methods and has led to the emergence of new powerful agents in the fields of optimal control [59], imitation learning [104] and reinforcement learning [31, 52]. Furthermore, it has emphasized the power of systems that operate based on first principles - such as learning and planning - through the accumulation of experiential data, and how by minimizing the prior knowledge encoded by human designers and by exploiting the ever growing availability of computational resources, we can unshackle our agents from the limitations of human cognition, and arrive at better solutions.

Acknowledgements

Much of the work presented in this document has been the result of my collaboration with many people within Deepmind. My supervisor David Silver has played a critical role in all of the work presented here. He has always provided me with guidance, inspiration and invaluable insight. I want to also thank my secondary supervisor Thore Graepel for his constructive suggestions and guidance. The *AlphaZero* algorithm presented in this work was designed in collaboration with David Silver, Julian Schrittwieser, Thomas Hubert, Karen Simonyan and Arthur Guez. David Silver led the project and designed the original reinforcement learning algorithm. Karen Simonyan proposed the idea of using a single network for representing the value and policy in *AlphaZero*. The software used both in *AlphaZero* and *MuZero* was implemented in collaboration with Julian Schrittwieser and Thomas Hubert. The initial version of the implicit model in *MuZero* was designed by Julian Schrittwieser. I developed the *Stochastic MuZero* algorithm and software with invaluable help and guidance provided by Julian Schrittwieser, Sherjil Ozair and Thomas Hubert.

To my beloved mother who never saw this adventure to the end.

Contents

1 Introduction **27**

1.1 Learning and Planning . . . 27
1.2 Reinforcement Learning . . . 28
1.2.1 Value function . . . 28
1.2.2 Policy . . . 28
1.2.3 Transition model . . . 29
1.3 Deep Learning in Reinforcement Learning . . . 29
1.4 Tree-based planning . . . 30
1.5 From *AlphaGo* to *Stochastic MuZero* . . . 30
1.5.1 Limitations of *AlphaGo* . . . 31
1.5.2 *AlphaZero* . . . 32
1.5.3 *MuZero* . . . 34
1.5.4 *Stochastic MuZero* . . . 35
1.6 Overview . . . 35

I Prior Work **37**

2 Reinforcement Learning **38**

2.1 Markov Decision Processes . . . 38
2.2 Policies and Value functions . . . 39
2.3 Value-Based methods . . . 40
2.3.1 Monte Carlo Methods . . . 40
2.3.2 Temporal Difference Learning . . . 41

2.4 Policy Gradient methods . . . 42
2.5 Model-Based methods . . . 43
2.5.1 Model learning . . . 43
2.5.2 Planning . . . 45
2.6 Search . . . 46
2.6.1 Heuristic Search . . . 46
2.6.2 Monte Carlo Tree Search . . . 47

3 Games for Reinforcement Learning 51
3.1 Board games . . . 51
3.1.1 Go . . . 51
3.1.2 Chess . . . 52
3.1.3 Shogi . . . 54
3.1.4 Backgammon . . . 54
3.2 Video games . . . 55
3.2.1 Atari . . . 56
3.2.2 2048 . . . 57

4 *AlphaGo* 59
4.1 Introduction . . . 59
4.2 Algorithm . . . 60
4.2.1 Networks . . . 60
4.2.2 Tree Search . . . 61
4.3 Limitations . . . 63

II Tree Search Planning with Deep Networks 64

5 *AlphaZero* 65
5.1 Introduction . . . 66
5.2 Algorithm . . . 67
5.2.1 Network . . . 67

5.2.2 Search . . . 67
5.2.3 Self-play . . . 68
5.2.4 Training . . . 70
5.3 Experiments . . . 70
5.3.1 *AlphaGo Zero* . . . 71
5.3.2 Results . . . 79
5.3.3 Ablations . . . 83
5.4 Conclusions . . . 85

6 *MuZero* **86**
6.1 Introduction . . . 86
6.2 Algorithm . . . 89
6.2.1 Model . . . 89
6.2.2 Search . . . 91
6.2.3 Self-play . . . 92
6.2.4 Training . . . 92
6.2.5 Reanalyze . . . 93
6.3 Experiments . . . 93
6.3.1 Results . . . 93
6.3.2 Ablations . . . 96
6.4 Conclusions . . . 99

7 *Stochastic MuZero* **100**
7.1 Introduction . . . 100
7.2 Algorithm . . . 101
7.2.1 Model . . . 101
7.2.2 Search . . . 105
7.2.3 Training . . . 106
7.3 Experiments . . . 106
7.3.1 Results . . . 106
7.3.2 Reproducibility . . . 109

7.3.3 Ablations 111
7.4 Conclusions 112

8 Conclusions 113
8.1 Open Problems in Learning and Planning 113
8.2 The promise of Learning and Planning 114

A *AlphaZero* Appendix 116
A.1 Domain knowledge in *AlphaZero* 116
A.2 Experimental Setup 117
A.2.1 Network Input Representation 117
A.2.2 Network Architecture 119
A.2.3 Configuration 120
A.2.4 Opponents 120
A.2.5 Match conditions 121
A.2.6 Elo ratings 122
A.3 Chess Openings 123

B *MuZero* Appendix 125
B.1 Hyperparameters 125
B.2 Data Generation 125
B.3 Network Input 126
B.3.1 Representation Function 126
B.3.2 Dynamics Function 127
B.4 Network Architecture 127
B.5 Training 129
B.6 *MuZero* Equations 130

Bibliography 132

List of Figures

2.1 **A Monte-Carlo Tree Search simulation**. Each simulation is comprised of four phases: *selection*, *expansion*, *rollout* and *backup*. During the selection phase the tree is traversed starting from the root node until a leaf edge is reached. The edges inside the tree are selected by computing the pUCT [99] formula. In the expansion phase a new node along with all its edges is added to the tree. A value estimate for the newly added node is computed during the rollout phase, by running games of selfplay starting at the current position until the end of the game and selecting actions using a rollout policy $\pi^{rollout}$. Finally, the statistics of the affected sub-tree are adjusted based on the value of the new node. 49

4.1 **The policy and value networks of *AlphaGo* A** The policy network of *AlphaGo* receives a board position s as input and generates a distribution over possible actions. **B** The value network of *AlphaGo* evaluates a board position s and returns the expected game outcome for the current player. 62

5.1 **Monte-Carlo tree search in *AlphaZero* a** During the selection phase, starting from the root node, the tree is traversed by selecting edges using the pUCT formula [99] (see 2.6.2) until a leaf node is reached. The pUCT formula combines the current value estimate for the edge Q with an exploration bonus term U which depends on the stored prior probability P and the visit count N of the edge. **b** At the expansion phase, a new node is added to the tree and the associated position s is evaluated by the neural network $(P(s,\cdot),V(s)) = f_\theta(s)$. **c** At the end of each simulation, the value estimates of the tree edges are updated to track the mean of all evaluations V in their corresponding subtree. **d** Once the search is complete, search probabilities π are returned, proportional to $N^{1/\tau}$, where N is the visit count of each move from the root state and τ is a parameter controlling temperature. 68

5.2 **Self-play reinforcement learning in *AlphaZero* a** *AlphaZero* generates a game of selfplay by executing a MCTS search at each step using the latest neural network f_θ. The actions are selected based on the search probabilities produced by the search. **b** *AlphaZero*'s neural network takes the current board position s as an input and outputs a vector $\mathbf{p}$ of move probabilities and a scalar value v which represents the probability of the current player winning the game starting at position s. The neural network parameters θ are updated so as to maximise the similarity of the policy vector $\mathbf{p}$ to the search probabilities π^{MCTS} , and to minimise the error between the value prediction v and the game outcome. The new parameters are used in the next iteration of self-play a. 69

5.3 **Empirical evaluation of *AlphaGo Zero*** **a** Performance of self-play reinforcement learning. The plot shows the performance of each MCTS player π^{MCTS} from each iteration i of reinforcement learning in *AlphaGo Zero*. Elo ratings were computed from evaluation games between different players, using 0.4 seconds of thinking time per move. For comparison, a similar player trained by supervised learning from human data, using the KGS data-set, is also shown. **b** Prediction accuracy on human professional moves. The plot shows the accuracy of the neural network f_θ, at each iteration of self-play i, in predicting human professional moves from the GoKifu data-set. The accuracy measures the percentage of positions in which the neural network assigns the highest probability to the human move. The accuracy of a neural network trained by supervised learning is also shown. **c** Mean-squared error (MSE) on human professional game outcomes. The plot shows the MSE of the neural network f_θ, at each iteration of self-play i, in predicting the outcome of human professional games from the GoKifu data-set. The MSE is between the actual outcome $z \in \{-1,+1\}$ and the neural network value v, scaled by a factor of $1/4$ to the range $[0,1]$. The MSE of a neural network trained by supervised learning is also shown. 72

5.4 **Performance of *AlphaGo Zero*.** **a** Learning curve for *AlphaGo Zero* using a larger 40 block residual network over 40 days. The plot shows the performance of each player π^{MCTS} from each iteration i of our reinforcement learning algorithm. Elo ratings were computed from evaluation games between different players, using 0.4 seconds per search. **b** Final performance of *AlphaGo Zero*. *AlphaGo Zero* was trained for 40 days using a 40 residual block neural network. The plot shows the results of a tournament between: *AlphaGo Zero*, *AlphaGo Master* (defeated top human professionals 60-0 in online games), *AlphaGo Lee* (defeated Lee Sedol), *AlphaGo Lee* (defeated Fan Hui), as well as previous Go programs Crazy Stone, Pachi and GnuGo. Each program was given 5 seconds of thinking time per move. *AlphaGo Zero* and *AlphaGo Master* played on a single machine on the Google Cloud; *AlphaGo Fan* and *AlphaGo Lee* were distributed over many machines. The raw neural network from *AlphaGo Zero* is also included, which directly selects the move a with maximum probability, without using MCTS. Programs were evaluated on an Elo scale [27]: a 200 point gap corresponds to a 75% probability of winning. 74

5.5 **Comparison of neural network architectures in *AlphaGo Zero* and *AlphaGo Lee*.** Comparison of neural network architectures using either separate (”sep”) or combined policy and value networks (”dual”), and using either convolutional (”conv”) or residual networks (”res”). The combinations ”dual-res” and ”sep-conv” correspond to the neural network architectures used in *AlphaGo Zero* and *AlphaGo Lee* respectively. Each network was trained on a fixed data-set generated by a previous run of *AlphaGo Zero*. **a** Each trained network was combined with *AlphaGo Zero*’s search to obtain a different player. Elo ratings were computed from evaluation games between these different players, using 5 seconds of thinking time per move. **b** Prediction accuracy on human professional moves (from the GoKifu data-set) for each network architecture. **c** Mean-squared error on human professional game outcomes (from the GoKifu data-set) for each network architecture. 76

5.6 **Go knowledge learned by *AlphaGo Zero*.** **a** Five human joseki (common corner sequences) discovered during *AlphaGo Zero* training. The associated timestamps indicate the first time each sequence occured (taking account of rotation and reflection) during self-play training. **b** Five joseki favoured at different stages of self-play training. Each displayed corner sequence was played with the greatest frequency, among all corner sequences, during an iteration of self-play training. The timestamp of that iteration is indicated on the timeline. At 10 hours a weak corner move was preferred. At 47 hours the 3-3 invasion was most frequently played. This joseki is also common in human professional play; however *AlphaGo Zero* later discovered and preferred a new variation. **c** The first 80 moves of three self-play games that were played at different stages of training, using 1600 simulations (around 0.4s) per search. At 3 hours, the game focuses greedily on capturing stones, much like a human beginner. At 19 hours, the game exhibits the fundamentals of life-and-death, influence and territory. At 70 hours, the game is beautifully balanced, involving multiple battles and a complicated ko fight, eventually resolving into a half-point win for white. 78

5.7 **Training *AlphaZero* for 700,000 steps.** Elo ratings were computed from games between different players where each player was given one second per move. **(A)** Performance of *AlphaZero* in chess, compared with the 2016 TCEC world-champion program *Stockfish*. **(B)** Performance of *AlphaZero* in shogi, compared with the 2017 CSA world-champion program *Elmo*. **(C)** Performance of *AlphaZero* in Go, compared with *AlphaGo Lee* and *AlphaGo Zero* (20 blocks over 3 days). 79

5.8 **Comparison with specialized programs. (A)** Tournament evaluation of *AlphaZero* in chess, shogi, and Go in matches against respectively *Stockfish*, *Elmo* and of *AlphaGo Zero* that was trained for 3 days. In the top bar, *AlphaZero* plays white; in the bottom bar *AlphaZero* plays black. Each bar shows the results from *AlphaZero*'s perspective: win ('W', green), draw ('D', grey), loss ('L', red). **(B)** Scalability of *AlphaZero* with thinking time, compared to *Stockfish* and *Elmo*. *Stockfish* and *Elmo* always receive full time (3 hours per game plus 15 seconds per move), time for *AlphaZero* is scaled down as indicated. **(C)** Extra evaluations of *AlphaZero* in chess against the most recent version of *Stockfish* at the time of writing, and against *Stockfish* with a strong opening book. Extra evaluations of *AlphaZero* in shogi were carried out against another strong shogi program *Aperyqhapaq* at full time controls and against *Elmo* under 2017 CSA world championship time controls (10 minutes per game plus 10 seconds per move). **(D)** Average result of chess matches starting from different opening positions: either common human positions, or the 2016 TCEC world championship opening positions. Average result of shogi matches starting from common human positions. CSA world championship games start from the initial board position. Match conditions are provided in appendix A. 82

5.9 **Repeatability of *AlphaZero* training on the game of chess.** The figure shows 6 separate training runs of 400,000 steps (approximately 4 hours each). Elo ratings were computed from a tournament between baseline players and *AlphaZero* players at different stages of training. *AlphaZero* players were given 800 simulations per move. Similar repeatability was observed in shogi and Go. . . . 83

5.10 **Learning curves showing the Elo performance during training in Go.** Comparison between *AlphaZero*, a version of *AlphaZero* that exploits knowledge of symmetries in a similar manner to *AlphaGo Zero*, and the previously published *AlphaGo Zero*. *AlphaZero* generates approximately $1/8$ as many positions per training step, and therefore uses eight times more wall clock time, than the symmetry-augmented algorithms. 84

1 **Planning, acting, and training with a learned model.** **(A)** How *MuZero* uses its model to plan. The model consists of three connected components for representation, dynamics and prediction. Given a previous hidden state s^{k-1} and a candidate action a^k, the *dynamics* function g produces an immediate reward r^k and a new hidden state s^k. The policy p^k and value function v^k are computed from the hidden state s^k by a *prediction* function f. The initial hidden state s^0 is obtained by passing the past observations (e.g. the Go board or Atari screen) into a *representation* function h. **(B)** How *MuZero* acts in the environment. A Monte-Carlo Tree Search is performed at each timestep t, as described in A. An action a_{t+1} is sampled from the search policy π_t, which is proportional to the visit count for each action from the root node. The environment receives the action and generates a new observation o_{t+1} and reward u_{t+1}. At the end of the episode the trajectory data is stored into a replay buffer. **(C)** How *MuZero* trains its model. A trajectory is sampled from the replay buffer. For the initial step, the representation function h receives as input the past observations $o_1, ..., o_t$ from the selected trajectory. The model is subsequently unrolled recurrently for K steps. At each step k, the dynamics function g receives as input the hidden state s^{k-1} from the previous step and the real action a_{t+k}. The parameters of the representation, dynamics and prediction functions are jointly trained, end-to-end by backpropagation-through-time, to predict three quantities: the policy $p^k \approx \pi_{t+k}$, value function $v^k \approx z_{t+k}$, and reward $r^k \approx u_{t+k}$, where z_{t+k} is a sample return: either the final reward (board games) or n-step return (Atari). 90

2 **Evaluation of *MuZero* throughout training in chess, shogi, Go and Atari.** The x-axis shows millions of training steps. For chess, shogi and Go, the y-axis shows Elo rating, established by playing games against *AlphaZero* using 800 simulations per move for both players. *MuZero*'s Elo is indicated by the blue line, *AlphaZero*'s Elo by the horizontal orange line. For Atari, mean (full line) and median (dashed line) human normalized scores across all 57 games are shown on the y-axis. The scores for R2D2 [68], (the previous state of the art in this domain, based on model-free RL) are indicated by the horizontal orange lines. Performance in Atari was evaluated using 50 simulations every fourth time-step, and then repeating the chosen action four times, as in prior work [84]. 94

3 **Evaluations of *MuZero* on Go (A), all 57 Atari Games (B) and Ms. Pacman (C-D)**. (**A**) Scaling with search time per move in Go, comparing the learned model with the ground truth simulator. Both networks were trained at 800 simulations per search, equivalent to 0.1 seconds per search. Remarkably, the learned model is able to scale well to up to two orders of magnitude longer searches than seen during training. (**B**) Scaling of final human normalized mean score in Atari with the number of simulations per search. The network was trained at 50 simulations per search. Dark line indicates mean score, shaded regions indicate 25th to 75th and 5th to 95th percentiles. The learned model's performance increases up to 100 simulations per search. Beyond, even when scaling to much longer searches than during training, the learned model's performance remains stable and only decreases slightly. This contrasts with the much better scaling in Go (A), presumably due to greater model inaccuracy in Atari than Go. (**C**) Comparison of MCTS based training with Q-learning in the *MuZero* framework on Ms. Pacman, keeping network size and amount of training constant. The state of the art Q-Learning algorithm R2D2 is shown as a baseline. Our Q-Learning implementation reaches the same final score as R2D2, but improves slower and results in much lower final performance compared to MCTS based training. (**D**) Different networks trained at different numbers of simulations per move, but all evaluated at 50 simulations per move. Networks trained with more simulations per move improve faster, consistent with ablation (B), where the policy improvement is larger when using more simulations per move. Surprisingly, *MuZero* can learn effectively even when training with less simulations per move than are enough to cover all 8 possible actions in Ms. Pacman. 97

4 **Details of *MuZero* evaluations (A-B) and policy improvement ablations (C-D)**. (**A-B**) Distribution of evaluation depth in the search tree for the learned model for the evaluations in Figure 3A-B. The network was trained over 5 hypothetical steps, as indicated by the red line. Dark blue line indicates median depth from the root, dark shaded region shows 25th to 75th percentile, light shaded region shows 5th to 95th percentile. (**C**) Policy improvement in Ms. Pacman - a single network was trained at 50 simulations per search and is evaluated at different numbers of simulations per search, including playing according to the argmax of the raw policy network. The policy improvement effect of the search over the raw policy network is clearly visible throughout training. This consistent gap between the performance with and without search highlights the policy improvement that *MuZero* exploits, by continually updating towards the improved policy, to efficiently progress towards the optimal policy. (**D**) Policy improvement in Go - a single network was trained at 800 simulations per search and is evaluated at different numbers of simulations per search. In Go, the playing strength improvement from longer searches is much larger than in Ms. Pacman and persists throughout training, consistent with our previous results. This suggests, as might intuitively be expected, that the benefit of models is greatest in precision planning domains. 98

1 ***Stochastic MuZero***. **(A)** Monte Carlo Tree Search used in *Stochastic MuZero*, where diamond nodes represent *chance* nodes and circular nodes represent *decision* nodes. During the *selection* phase edges are selected by applying the pUCT formula in the case of decision nodes, and by sampling the prior σ in the case of chance nodes. **(B)** Training of stochastic model in *Stochastic MuZero*. Here for a given trajectory of length 2 with observations $o_{\leq t:t+2}$, actions $a_{t:t+2}$, value targets $z_{t:t+2}$, policy targets $\pi_{t:t+2}$ and rewards $u_{t+1:t+K}$, the model is unrolled for 2 steps. During the unroll, the encoder e receives the observation $o_{\leq t+k}$ as an input and generates a chance code c_{t+k} deterministically. The policy, value and reward outputs of the model are trained towards the targets π_{t+k}, z_{t+k} and u_{t+k} respectively. The distributions σ^k over future codes are trained to predict the code produced by the encoder. 102

2 **Planning in *2048***. a) *Stochastic MuZero*, trained using 100 simulations of planning with a learned stochastic model, matched the performance of *AlphaZero*, using 100 simulations of a perfect stochastic simulator, while a deterministic learned model (*MuZero*) performed poorly. b) Evaluation of final agent using different levels of search. *Stochastic MuZero* scales well during evaluation to intermediate levels of search (roughly comparable to 3-ply lookahead), exceeding the playing strength of the state-of-the-art baseline [63]. However, as the number of simulations increases we observe diminishing returns due to imperfections of the learned model. 107

3 ***Stochastic MuZero* in Backgammon**. a) *Stochastic MuZero*, trained using 1600 simulations of planning with a learned stochastic model, matched the performance of *AlphaZero*, trained using 1600 simulations of a perfect stochastic simulator, as well as matching the superhuman-level program GNUbg Grandmaster. A deterministic learned model (*MuZero*) performed poorly. b) *Stochastic MuZero*'s model scaled well to large searches, and exceeded the playing strength of GNUbg Grandmaster when using more than 10^3 simulations. 108

4 ***Stochastic MuZero* in Go**. Comparison of *Stochastic MuZero* and *MuZero* in the game of Go. a) *Stochastic MuZero* and *MuZero* when compared in 9x9 Go. *MuZero* has a search budget of 200 simulations during training of 800 during evaluation, while *Stochastic MuZero* uses 400 simulations during training and 1600 during evaluation. The Elo scale was anchored so that the performance of the final *MuZero* baseline corresponded to an Elo of 2000. b) *Stochastic MuZero* and *MuZero* when compared in 19x19 Go. *MuZero* has a search budget of 400 simulations during training of 800 during evaluation, while *Stochastic MuZero* uses 800 simulations during training and 1600 during evaluation. The Elo scale was anchored so that the performance of the final *MuZero* baseline corresponded to an Elo of 2000. 110

5 ***Stochastic MuZero* reproducibility across all domains.** We ran our method *Stochastic MuZero* in all environments using 9 different seeds to measure its robustness to random initialization. We observed that there is minimal variation in the performance of *Stochastic MuZero* for all different seeds. Due to the computational cost of each experiment we used a smaller number of training steps for each experiment. 111

6 **Average distribution of learned chance outcomes.** The average distribution of learned chance outcomes over all chance nodes after running *Stochastic MuZero* at each game for 5 episodes. 112

1 **Matches starting from the most popular human openings**. *AlphaZero* plays against **(A)** *Stockfish* in chess and **(B)** *Elmo* in shogi. In the left bar, *AlphaZero* plays white, starting from the given position; in the right bar *AlphaZero* plays black. Each bar shows the results from *AlphaZero*'s perspective: win (green), draw (grey), loss (red). The percentage frequency of self-play training games in which this opening was selected by *AlphaZero* is plotted against the duration of training, in hours. 124

1 **Equations summarising the *MuZero* algorithm**. Here, $\phi(x)$ refers to the representation of a real number x through a linear combination of its adjacent integers, as described in the Network Architecture section. 131

Chapter 1

Introduction

This thesis studies the problem of planning and learning in complex environments for artificial intelligent systems.

1.1 Learning and Planning

The goal of artificial intelligence is to devise algorithms which exhibit intelligent behaviour. We can define intelligence as the ability for a system to interact with its environment and make decisions to achieve goals [76]. Two features of paramount importance for intelligence, as demonstrated by biological systems such as, humans and animals, are the ability to learn and to plan. Learning is the process by which a system improves its decision-making behaviour based on previous interactions it had with its environment. On the other hand, during planning an intelligent system considers possible future scenarios and adapts its behaviour based on those predictions. Learning and planning are closely related, since both attempt to analyse past or future experience to improve the decision-making abilities of the system at the present time.

Given their importance, constructing intelligent systems with learning and planning capabilities has long been one of the main challenges in the pursuit of artificial intelligence. In this work we will present new algorithms which combine learning and planning to achieve super-human performance in a wide range of complex environments and will demonstrate how these two processes can bootstrap from each other. Our methods build upon the latest developments in reinforcement learning,

deep learning and tree-based planning.

1.2 Reinforcement Learning

Reinforcement learning (RL) is a field of research within artificial intelligence, which studies the problem of making optimal decisions in complex systems. Under this formalism, we consider two main subsystems: the *agent* and the *environment*. The environment implements the dynamics of the system and defines the optimality condition via a scalar reward signal, which is used to either encourage or punish the behaviour exhibited by the agent. On the other hand, the agent perceives the state of the environment and makes decisions, with the goal of maximizing the cumulative reward it receives by the environment. The decisions the agent can make at any state are described by a set of actions available to it. A simple example of a reinforcement learning problem is that of a robot navigating a room and trying to exit from it. In that scenario, the robot is the agent that perceives the state of the real world through its sensors and takes actions by controlling its motors. Subsequently, we can define a reward signal that gives negative reward for bumping into a wall and a positive reward for approaching the door. The environment in this simple example is the real world, the system that produces the reward signal and the sensors on the robot.

1.2.1 Value function

The goal of an agent is to maximize the cumulative reward it receives from the environment. A *value function* computes the expected cumulative reward that the agent will receive starting from any state in the environment and following its current decision making behaviour. Given this definition, the reinforcement learning problem can be formulated as finding the behaviour that corresponds to the maximum value function.

1.2.2 Policy

A *policy* is a function used to mathematically describe the decision-making behaviour of the agent. The policy examines the current state of the environment and generates a probability distribution over the possible actions that the agent can

make at this state. The optimal policy is the one that leads to the highest cumulative reward, or equivalently, maximum value function.

1.2.3 Transition model

A *transition model* is a function that approximates the dynamics of the environment. It receives the current state of the environment and an action as inputs and produces the next state and intermediate reward. These models are constructed by the agent and can be used for planning and learning. By using a transition model the agent can evaluate and improve its policy without directly interacting with the environment.

1.3 Deep Learning in Reinforcement Learning

Artificial neural networks are biologically inspired computational systems [82], which represent a class of universal function approximators [56]. They consist of groups of neurons, each of which is a simple computational node. A neuron receives a set of scalar inputs, combines them in a linear fashion and applies a non linear function to the resulting value. Neurons are grouped into layers, where all neurons in a layer operate on the same inputs. Subsequently, these layers can be stacked together to produce deeper architectures. In recent years the development of improved hardware and novel algorithms has led to an explosion in the use of deep neural networks in a wide range of tasks in artificial intelligence, such as machine translation [130, 114, 23], image recognition [71, 118, 119], natural language processing [18], and reinforcement learning [84, 83, 101, 36].

Deep reinforcement learning refers to the study of the intersection of reinforcement learning and deep neural networks. In deep reinforcement learning neural networks are used to represent value functions, policies and transition models. This has enabled the applicability of reinforcement learning methods to an ever expanding list of complex domains such as board and video games [122, 124, 84, 36, 83], medicine [77], scientific discoveries [105, 30] and robotics [94].

1.4 Tree-based planning

In the context of reinforcement learning, planning is the process by which an agent makes use of its internal transition model to evaluate and improve its policy. Depending on the task at hand, the agent could have access to a perfect model of the real environment dynamics or it might need to construct one using the experience it has collected by interacting with the environment.
Tree-based planning algorithms solve the problem of planning by constructing trees of possible future trajectories. These methods were first proposed in the context of game theory [133], in an effort to construct intelligent game-playing agents. To better illustrate tree-based planning we can consider the game of chess, where the goal of the agent is to beat its opponent. Starting from any state in the game, the agent tries to identify the action which maximizes their chances of winning the game. In tree based planning the agent constructs a tree by considering all of their actions and subsequently all of their opponent's actions repeatedly. Given a tree an agent can select the action which maximizes their chance of winning by considering all possible game outcomes. In our chess example an agent can use the minimax [92] decision rule, which finds the action that maximizes the chance of the player winning conditioned on the fact that their opponent tries to minimize it. However, as more plies are considered the size of the tree grows exponentially. A number of practical algorithms to counter the above limitation have been proposed in the literature [20, 98], which attempt to limit the depth and the breadth of the tree using pruning heuristics, evaluation functions, and sampling approximations.

1.5 From *AlphaGo* to *Stochastic MuZero*

AlphaGo is arguably one of the most successful examples of this new generation of deep reinforcement learning algorithms. It was the first program to beat a human professional player in the game of Go. Go is a popular ancient two player board game, which despite its simple rules has proven extremely challenging for computer programs to master and it has been considered as a grand challenge in the field of artificial intelligence [107]. *AlphaGo* managed to achieve superhuman per-

formance, by leveraging the latest developments in the fields of deep learning and tree based planning. Specifically, it replaced the heuristic evaluation functions used by previous approaches [27] with deep neural networks trained via reinforcement learning, and adapted its tree-based search to effectively plan with them.

Despite its success, *AlphaGo* utilized domain specific knowledge and data and, withstanding significant human effort, it could only be applied to the game of Go. *AlphaGo*, nonetheless, proposed the main principles upon which a new class of general algorithms could be built, namely the use of deep neural networks trained using reinforcement learning techniques and combined with a general tree based planning algorithm, Monte Carlo Tree Search.

1.5.1 Limitations of *AlphaGo*

AlphaGo makes use of two neural networks when planning. A value function network which evaluates a Go board position and estimates the probability of winning for each player, and a policy network which when applied to the same position recommends a list of promising actions for either player. The policy network was trained in a supervised learning setting, where millions of human games were provided as training data, and the network was trained to imitate the behaviour of the human players. Subsequently, the performance of the network was further improved through *self-play* [1]. The resulting policy was used to generate new artificial games which then acted as training data to obtain a value function. During game play, *AlphaGo* combined those two networks with a Monte Carlo tree search and a heuristic evaluation function, which made use of hand-crafted rollouts [2]. Given the above description we can identify a number of limitations in the *AlphaGo* algorithm:

- It requires access to expert data. *AlphaGo* made use of an extended database of Go games player by human professionals to train its policy network. This

[1]Self-play is a common practice in reinforcement learning, and it involves training a agent such that it can beat previous versions of itself.

[2]A rollout is the process of repeatedly applying a policy starting from a game state until the end of the game. An evaluation function can be obtained by executing multiple rollouts and then computing the average outcome, see more in 2.6.2

limits its applicability to domains where abundant expert data are available.

- A number of specialized hand-crafted features were provided as inputs to the networks used by *AlphaGo*. This includes statistics and game information which is not readily available in the simple board representation.

- *AlphaGo*'s evaluation function combined the value estimates generated by its value function with the ones computed via hand-crafted rollouts which encoded a significant amount of prior human knowledge.

- Finally, during planning, *AlphaGo* had access to a perfect simulator of the game rules and dynamics. In many real world applications such a simulator is not available or is too computationally expensive.

As a result, extending the applicability of *AlphaGo* to new domains requires solutions which overcome the aforementioned limitations. In the following sections we will describe methods which remove these restrictions and result in new more powerful and general agents.

1.5.2 *AlphaZero*

AlphaZero is a general reinforcement learning algorithm which can be applied to any domain where a perfect simulator of the rules and dynamics is available. It follows the same principles as *AlphaGo*, namely it combines a Monte Carlo Tree search planning algorithm with deep neural networks. However, *AlphaZero* takes the idea of self-play a step further and uses this process to train both its value and policy networks completely from scratch without the need for any expert data, and without using any prior human knowledge and heuristics. The main idea is to use planning to support learning and conversely learning to improve planning. As was demonstrated by *AlphaGo*, combining a pre-trained policy and value with a Monte Carlo Tree Search produces a significantly improved policy. *AlphaZero* makes use of this property as a learning mechanism. Starting from randomly initialized networks, *AlphaZero* employs planning to generate an improved policy, which is then used to generate games through selfplay. The improved policy is then encoded

back into *AlphaZero*'s networks via supervised learning, where the policy network is trained to imitate the acting policy and the value network is trained to predict the observed game outcomes. This process is repeated until convergence. The main properties of the *AlphaZero* algorithm are summarized below:

- *AlphaZero* is trained completely from scratch using reinforcement learning and selfplay. It does not require access to expert data and it does not assume any prior human knowledge besides the rules of the game.
- Each position is evaluated solely by the value function and no hand-crafted domain specific rollouts are used inside the search. The value function was trained to predict the outcome of millions of games of selfplay.
- The inputs provided to the neural networks of *AlphaZero* are plain numeric descriptions of the board state and are not based on expert domain specific knowledge.
- Due to the above properties, the applicability of *AlphaZero* is not limited to a single domain but instead the same algorithm can be applied to different games without any changes. In the experiments described in this work, *AlphaZero* was tested in the games of Go, chess and shogi where it achieved superhuman performance while significantly outperforming previous algorithmic approaches.

The generality of *AlphaZero* has allowed its adoption by numerous research groups to tackle a wide range of challenging real world problems, such as in chemical synthesis [105] and quantum computing [30]. However, the scope of *AlphaZero* is still limited to environments for which a fast and accurate simulator of the dynamics is available both during training and deployment. Finally, the algorithm was originally only applied to two player zero-sum perfect information board games, instead of the more standard single player reinforcement learning setting.

1.5.3 *MuZero*

The promise of reinforcement learning lies within its ability to solve a wide range of problems, ranging from robotics to artificial personal assistants and beyond. In most domains, hand-coding the environment dynamics is either impossible or the resulting simulator is prohibitively expensive in terms of compute, rendering its use during deployment infeasible. As a result, applying the powerful principles of planning and learning demonstrated by *AlphaZero* requires methods that can obtain approximate models of the environment dynamics based only on the experience collected by the agent in the course of its interactions with the environment.

Model-based reinforcement learning studies the problem of obtaining models of the environment dynamics and then using them for planning and learning. Under the classic instantiation of this paradigm, the agent collects a set of state transitions by interacting with the environment, and subsequently uses them as training data to obtain a model of the dynamics. In principle this approach could be combined with *AlphaZero* without the need of any other changes to the algorithm. However, there has been a plethora of previous work demonstrating the limitations of this approach [66, 78]. The main issues are summarized below:

- *Model capacity*. In many complex environments learning an accurate model of the environment dynamics in the observation space can be challenging. This is the case especially in problems with high dimensionality pixel observations, which is common in many challenging problems in deep reinforcement learning (Atari [15], DM-Lab [13], OpenAI Gym [17] etc.).

- *Compounding errors*. Planning requires unrolling the learned model for many steps into the future. As a result even small errors introduced at each unroll step can easily accumulate to such a degree as to render the model useless.

- *Background noise*. In many environments there are features in the observation space which are irrelevant to the task at hand, but could limit the capacity of the learned transition model. For example the natural lighting in a robotics problem is irrelevant to the task itself but it affects the observations the robot

receives through its cameras.

MuZero addresses the above limitations by modeling only the quantities which are useful during planning, namely the policy, value and reward functions. This ensures that the model makes better use of its capacity and that it learns to ignore any irrelevant background noise.

1.5.4 *Stochastic MuZero*

Real world environments tend to be messy and hard to model. In many cases the transition dynamics are affected by factors outside the control of the agent itself. As a result applying the same action to the same state can lead to radically different next states. For example, there are environments that involve explicit stochastic events such as a dice roll, or events that are perceived as stochastic by the agent such as the wind conditions in a robotics task or the action selected by a different agent in a multi-agent setting. *MuZero* explicitly assumes that the environment dynamics are deterministic and are affected only by the actions it selects. However, a more general and theoretically sound approach should consider a distribution over possible future events and construct plans accordingly.
In this work, in order to address the above limitation, we propose a new agent *Stochastic MuZero* that explicitly models the distribution of possible future events. It directly extends *MuZero* to a wide range of new stochastic domains, while matching its performance in deterministic ones. It achieves this by using *afterstates*, in the model learning and planning algorithms. An afterstate is an intermediate state after an action has been applied but before the environment has transitioned to an actual next state. This way the contributions of the actions selected by the agent and of the environment stochasticity can be modelled separately.

1.6 Overview

In the first part of this thesis, we provide a survey of the relevant literature review.

- In Chapter 2, we describe the key concepts in the reinforcement learning framework and we survey the relevant work in model-based reinforcement learning.

- in Chapter 3, we provide a short description of *AlphaGo*, the algorithm which constituted our main inspiration for developing our new methods.
- in Chapter 4, we examine the environments which were used as testbeds in this work and we review some notable previous approaches which were used to tackle them.

In the second part of this work we present our contributions to the field of model based reinforcement learning. Specifically,

- In Chapter 6, we present *AlphaZero*, a general reinforcement learning algorithm which we apply to the board games of Go, chess and Shogi.
- In Chapter 7, we introduce *MuZero*, a general model-based reinforcement learning algorithm, which extends *AlphaZero* to single and two player deterministic environments for which we do not assume access to a perfect simulator of the environment dynamics.
- In Chapter 8, we describe *Stochastic MuZero* which further extends the applicability of the *MuZero* agent to stochastic environments.
- In Chapter 9, we discuss the conclusions of this work and provide insights into possible future research directions.

Part I

Prior Work

Chapter 2

Reinforcement Learning

Reinforcement learning (RL) studies the problem of optimal decision making in sequential processes. The RL problem is formulated in terms of an *agent* interacting with an *environment* in discrete time steps. The agent is the entity responsible for making decisions, while everything else is considered part of the environment. At each time step t, the agent receives an observation s_t from the environment and applies an action a_t. Subsequently, the environment generates a scalar reward signal r_t and transitions to a new state s_{t+1}. The goal of the agent is to select actions in such a way as to maximize the cumulative reward $\sum_t^\infty r_t$, given the history of past states, actions and rewards $h_t = \{s_0, a_0, r_1, s_1, a_1 \cdots s_t\}$ it has observed.

2.1 Markov Decision Processes

All environments in RL are fully described by their transition probabilities $Pr(s_{t+1}, r_{t+1} \mid h_t, a_t)$. Markov Decision Processes (MDPs) constitute a particular subset of environments, which satisfy the Markov property:

$$Pr(s_{t+1}, r_{t+1} \mid h_t, a_t) = Pr(s_{t+1}, r_{t+1} \mid s_t, a_t) \tag{2.1}$$

MDPs have been heavily studied in the field of RL [117] due to their mathematical simplicity and their wide applicability. We call an MDP fully observable if it provides its Markov state s_t as an observation to the agent. On the other hand, partially observable MDPs do not expose their true state but rather an observation which directly depends on it. In deterministic MDPs, the transition probability function is

described by a Dirac delta function, where for any given state s_t and action a_t the MDP always transitions to the same state s_{t+1}.

2.2 Policies and Value functions

The behaviour of an agent is controlled by its *policy* π. The policy is a function that maps each state $s_t \in \mathcal{S}$ to a probability distribution over actions $a \in \mathcal{A}$:

$$\pi(a_t \mid s_t) = Pr(A_t = a_t \mid S_t = s_t) \tag{2.2}$$

The goal of an agent is to find a policy that maximises the accumulated reward it receives from its environment. We can quantify the performance of a policy using its corresponding *value function*. The value function is defined as the expected discounted sum of rewards the agent will receive from the environment, when starting from any state s_t and selects actions using its policy π:

$$V^{\pi}(s_t) = \mathbb{E}_{\pi}\left[\sum_{\tau=0}^{+\infty} \gamma^{\tau} r_{t+\tau} \,\middle|\, S_t = s_t\right] \tag{2.3}$$

We can use the value function to compare different policies. A policy π' is better than a policy π, $\pi' \geq \pi$ when the value function of π' is greater than the one for π for all states $V^{\pi'}(s_t) \geq V^{\pi}(s_t)$. The policy that achieves the maximum discounted sum of rewards starting from any state s_t is called the *optimal policy* and is denoted with π^*. In the general case, there can be multiple optimal policies, however, they all share the same *optimal value* function $V^*(s_t)$.

In many cases, it is useful to consider the action value function $Q(s_t, a_t)$. The action value function is a value function which is conditioned both on the current state s_t and action a_t:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{\pi}\left[\sum_{\tau=0}^{+\infty} \gamma^{\tau} r_{t+\tau+1} \,\middle|\, S_t = s_t, A_t = a_t\right] \tag{2.4}$$

The action-value function represents the expected discounted sum of rewards the agent will receive if they select an action a_t at time t and follow the policy thereafter. From the definition it follows that action-value and value functions are connected by the following equation:

$$Q^{\pi}(s_t, a_t) = \sum_{s_{t+1}, r_{t+1}} Pr(s_{t+1}, r_{t+1} \mid s_t, a_t)\left[r_{t+1} + \gamma V^{\pi}(s_{t+1})\right] \tag{2.5}$$

The action value function provides a direct connection between values and selected actions. This is extremely useful, since finding an optimal action value function Q^* allows us to easily obtain an optimal policy π^* by greedily selecting actions under the action value function, $\pi^* = \arg\max_a Q^*(s,a)$.

2.3 Value-Based methods

Value-based methods make use of the value function to improve the policy of the agent. Given a value function $V(s_t)$ we can obtain an improved policy π' by greedily selecting actions under $V(s_t)$:

$$\pi' = \arg\max_a \sum_{s_{t+1}, r_{t+1}} Pr(s_{t+1}, r_{t+1} \mid s_t, a_t)\left[r_{t+1} + \gamma V^{\pi}(s_{t+1})\right] \tag{2.6}$$

If the above process was repeated for a number of steps, by first computing a value function V corresponding to a policy π and subsequently greedily obtaining a new policy π', the final value function and policy would be optimal [117].

2.3.1 Monte Carlo Methods

According to the definition of the value function, computing it in a closed form requires knowledge of the environment transition dynamics. Monte Carlo methods, instead, obtain unbiased estimates of the value function using only sampled trajectories collected by the RL agent. Given an trajectory of real experience with observed rewards $(r_0, \cdots, r_T)$ and a discount factor $\gamma \in [0,1]$, the Monte Carlo return estimate for each state s_t is given by:

$$G_t = \sum_{\tau=t}^{T} \gamma^{\tau-t} r_{\tau+1} = r_{t+1} + \gamma r_{t+1} + \cdots + \gamma^{T-t-1} r_T \tag{2.7}$$

Subsequently, a value function estimate can be computed by averaging multiple such estimates:

$$\hat{V}(s_t) = \frac{1}{n} \sum_i G_t^{(i)} \tag{2.8}$$

The above Monte Carlo estimator is bias free, but it suffers from high variance and it requires many samples to obtain an accurate estimate of the true value.

2.3.2 Temporal Difference Learning

A fundamental property of value functions is that they can be defined recursively through the *Bellman equation* [117]. This equation connects the value function of s_t with the value function of the subsequent states:

$$V^{\pi}(s_t) = \sum_{a_t \in \mathcal{A}} \pi(a_t \mid s_t) \sum_{s_{t+1}, r_{t+1}} Pr(s_{t+1}, r_{t+1} \mid s_t, a_t) \left[r_{t+1} + \gamma V^{\pi}(s_{t+1}) \right] \tag{2.9}$$

Computing the value function using the Bellman operator still requires access to the transition dynamics of the environment. However, we can easily obtain a Monte Carlo sampled based estimate:

$$\hat{V}^{\pi}(s_t) = \frac{1}{n} \sum_i \left[r_t^{(i)} + \gamma V^{\pi}(s_{t+1}^{(i)}) \right] \tag{2.10}$$

A problem with the above equation is that it assumes access to the true value function at the future states s_{t+1}. In practice, we use our current best estimate of the value at future states s_{t+1} to obtain a new estimate for the current state s_t. This is a common practice in statistics called *bootstrapping*, which reduces the variance in the estimates at the expense of introducing bias. In the field of value based RL, this approach is called *Temporal Difference Learning*, due to the temporal difference error (TD-error) which is defined as the difference between the current estimate of the value $\hat{V}(s_t)$ and the one step estimate $r_{t+1} + \gamma \hat{V}(s_{t+1})$:

$$\delta_t = r_{t+1} + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t) \tag{2.11}$$

By minimizing the above error for all states in the environment, we can obtain the true value function [117].

The above one-step TD-error has low variance but high bias, which, in practice, can significantly slow down learning. We can speed up learning by increasing the number of steps before bootstrapping at the expense of higher variance in our estimates. The n-step TD estimates can be computed as follows:

$$\begin{aligned} G_t^n &= r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^n \hat{V}(s_{t+n}) \\ \delta_t^n &= G_t^n - \hat{V}(s_t) \end{aligned} \tag{2.12}$$

From the above equation, it is obvious that by setting $n = T$ we obtain a Monte Carlo estimate for the value, while by setting $n = 1$ we are back to the one-step TD-error.

This way by varying n we can control the variance bias trade-off in our estimates. Another way to reduce the variance and have more control over the variance bias trade-off is to combine our n-step estimates into a single estimate. This approach is called TD(λ), and it combines all the n-step estimates using a $\lambda \in [0,1]$ parameter as follows:

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{T} \lambda^{n-1} G_t^n \tag{2.13}$$

By varying the value of λ, we can compute the one step TD-error for $\lambda = 0$ or the Monte Carlo estimate for $\lambda = T$.

2.4 Policy Gradient methods

Policy gradient methods solve the reinforcement learning problem by explicitly representing the agent's policy $\pi(s,a)$ with a parametric function $\pi_\theta(s,a)$ and subsequently, optimizing θ to maximize the agent's average reward per time-step:

$$J(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s,a) \mathcal{R}_s^a \tag{2.14}$$

where $d^{\pi_\theta}(s)$ denotes the stationary distribution of states of the MDP under the current policy π_θ. In these approaches θ is trained using gradient ascent by differentiating $J(\theta)$ using the policy gradient theorem [117]:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)\right] \tag{2.15}$$

Computing the above gradient requires obtaining an estimate of Q^{π_θ}. A variety of approaches have been proposed in the literature for estimating Q^{π_θ}. In the simplest case a Monte Carlo estimate can be used, as proposed in the REINRORCE algorithm [135]. On the other hand, actor-critic methods obtain lower variance estimate of the value by learning a separate value function (see 2.3). Here, there are two components, the actor which is responsible for training the policy and the critic which learns a value for the acting policy. Special care should be taken when the value function approximation is used to ensure that the critic and actor parameters are compatible: $\nabla_\phi Q_\pi(s,a) = \nabla_\theta \pi_\theta(s,a)$.

One of the main drawbacks of policy gradient methods is that they suffer from high variance in the gradient estimation, which makes them less efficient in com-

parison to the value-based approaches. This can be ameliorated by introducing a baseline function $b(s)$ which depends only on the state s, and replacing $Q^{\pi_\theta}(s,a)$ with $Q^{\pi_\theta}(s,a) - b(s)$. By setting the baseline to be equal to the value function $V^{\pi_\theta}(s)$, we can minimize the variance of the gradient estimate.

The fact that policy gradient methods can directly optimize a parameterized policy makes them advantageous in problems with a high dimensional or continuous action space, or in domains where a stochastic policy is preferable.

2.5 Model-Based methods

As we saw previously, all reinforcement learning environments are described by their transition probabilities $Pr(s_{t+1}, r_{t+1} \mid h_t, a_t)$. Model based methods attempt to learn an approximate model $\hat{Pr}(s_{t+1}, r_{t+1} \mid h_t, a_t)$ of the real transition probabilities and subsequently use it to improve the acting policy $\pi(a,s)$. From a theoretical standpoint, model-based methods can be more data efficient and lead to a stronger overall performance, by leveraging synthetic experience generated by the model instead of having to interact with the real environment. However, learning an accurate enough model of complex environments can be extremely challenging in practice [66, 78]. At the same time using synthetic data which are out of distribution can significantly hinder the policy improvement operators employed by model-based methods.

2.5.1 Model learning

Model learning considers the problem of learning an approximate model of the transition probabilities of an environment based on the real experience collected by an agent. This is an instance of a supervised learning problem where a function of parameters θ is fitted to the observed data. The recent success of deep neural networks has lead to their wide adoption for the problem of model learning, since they can model highly complex environment dynamics and generalize well to unseen data. In this work we will focus on models represented by deep neural networks.

There are two main categories of models proposed in the literature:

- *Observation models*, which approximate the environment dynamics at the

level of raw observations.

- *Agent state models*, which approximate the environment dynamics implicitly in terms of an internal representation of states.

In the following paragraphs, we describe each approach separately.

Observation models [87, 22, 66] operate at the level of raw observations. Given an observation o_t and an action a_t they return the next observation o_{t+1} and the intermediate reward r_t. Those models are trained end-to-end by fitting them to a dataset of observed transitions. Despite their simplicity, these models have a number of drawbacks:

- High computational cost, especially in the case of high dimensional observations.
- High error accumulation, since any small errors at the observation prediction quickly accumulate as the model is unrolled for multiple steps.
- Inefficiencies, since the model capacity can be wasted on background observation features which are irrelevant to the problem at hand.

The above issues make such models unconducive to planning.

Agent state models [87, 46, 51, 90] attempt to overcome the limitations of observation models by implicitly learning the environment dynamics. As we saw above, once a model is trained it is combined with an RL method or a planning algorithm so as to improve the final acting policy $\pi(s,a)$. Latent models exploit this fact by defining a latent space of environment states which can accurately predict only the quantities which are useful for the policy improvement step. In this framework, the model is conditioned on the current observation o_t and future actions $a_t, \cdots, a_{t+k}$ and is unrolled for k steps. Subsequently, it is trained to make predictions about rewards, values, policies or observations at each timestep based on the current latent state. This reduces the computational cost of the model since it removes the need for modelling high dimensional observations, it makes the model robust to error accumulation by unrolling the model for k steps during training and finally, it focuses

the capacity of the model only on features which matter for the policy improvement step.

A particularly interesting instantiation of latent models focuses on predicting the value function [121, 113, 38, 37, 39, 87] end-to-end. The main idea of these methods is to construct an abstract MDP model such that planning in the abstract MDP is equivalent to planning in the real environment. This equivalence is achieved by ensuring *value equivalence*, i.e. that, starting from the same real state, the cumulative reward of a trajectory through the abstract MDP matches the cumulative reward of a trajectory in the real environment. The predictron [113] introduced value equivalent models for value prediction. Although the underlying model still takes the form of an MDP, there is no requirement for its transition model to match real states in the environment. Instead the MDP model is viewed as a hidden layer of a deep neural network. The unrolled MDP is trained such that the expected cumulative sum of rewards matches the expected value with respect to the real environment, e.g. by temporal-difference learning.

Value equivalent models have also been applied to optimising value (with actions). Value-aware model learning [38, 37] constructs an MDP model, such that a step of value iteration using the model produces the same outcome as the real environment. TreeQN [39] learns an abstract MDP model, such that a tree search over that model (represented by a tree-structured neural network) approximates the optimal value function. Value iteration networks [121] learn a local MDP model, such that many steps of value iteration over that model (represented by a convolutional neural network) approximates the optimal value function. Value prediction networks [87] learn an MDP model grounded in real actions; the unrolled MDP is trained such that the cumulative sum of rewards, conditioned on the actual sequence of actions generated by a simple lookahead search, matches the real environment.

2.5.2 Planning

Planning refers to the process of improving the acting policy of the agent using a model of the environment instead of directly interacting with it. The main idea is to use simulated experience to estimate and improve the value and policy by evaluating

possible future trajectories starting from a given state s.

Model-based planning In model based planning, the main idea is to use the model as a substitute for the real environment and apply a reinforcement learning algorithm to find the optimal policy. Subsequently, the resulting policy can be applied directly to the environment. The performance of the obtained policy is directly affected by the quality of the trained model. In complex environments, where obtaining an accurate model can be an extremely challenging task, such methods can lead to poor performance. On the other hand, in domains with simple dynamics but complex optimal policies or value functions, this approach could allow for significant improvement in data efficiency.

Sample-based planning In sample-based planning, the model is used to generate artificial trajectories. These trajectories can then be used to estimate the value function and improve the acting policy. In this approach, the model is not required to accurately approximate the true transition probabilities $Pr(s_{t+1}, r_{t+1} \mid h_t, a_t)$, but rather it suffices that it can generate samples that follow the same distribution, for example using neural network based generative temporal models [46, 45, 44].

In the Dyna [116] paradigm, the synthetic data generated by the model are combined with real environment trajectories in order to augment the experience of the agent, speed up its learning and improve its data efficiency.

2.6 Search

Similarly to planning, search methods attempt to improve the policy of the agent by considering future trajectories starting from the current state s. They achieve this by constructing a tree of future paths given sequences of possible actions. The goal of the process is to find the sequence of actions that result in the highest cumulative reward.

2.6.1 Heuristic Search

Heuristic search is an umbrella term used to describe state space tree-based planning methods. Given a current state s it is possible to find the optimal sequence of actions by constructing a tree of all possible future trajectories until the end of the

game. However, this approach scales exponentially with the size of the game and the number of available actions at each internal node, and it is intractable even for small domains. Heuristic search methods use heuristics to prune the tree both width and depth wise. This class of algorithms assume access to an evaluation function, usually implemented using domain specific heuristics, which provides an estimate of the value function for each leaf node of the tree. The internal nodes are updated based on the value of their children using an appropriate backup operator. The evaluation function is used to reduce both the depth of the tree, since it provides an immediate feedback for non-terminal leaves, and its width since the value estimates can be used to prune uninteresting regions of the search space.

Many examples of popular search algorithms fall into this category such as A^* [47] search which uses a max backup operator, expectimax with a Bellman backup operator and alpha-beta [92] with min-max backups.

2.6.2 Monte Carlo Tree Search

Monte Carlo Tree Search(MCTS) is a general search algorithm that iteratively constructs a tree of possible future trajectories. The tree consists of nodes which correspond to states s and edges (s,a) which describe state action pairs. Each edge stores a set of statistics:

$$P(s,a), N(s,a), Q(s,a) \tag{2.16}$$

where $P(s,a)$ is a prior probability for the edge, $N(s,a)$ is the total times it was visited, and $Q(s,a)$ is its current value estimate. In some variants of MCTS, a uniform prior probability is used, where all possible edges starting at node s are considered equiprobable. The tree is constructed via a series of simulations, where each one of them is comprised of four phases selection, expansion, rollout and backup.

Selection During the selection phase the algorithm traverses the tree starting at the root, until it reaches a leaf node. For each intermediate node the algorithm selects an edge to traverse using a version of the UCT [69] formula.

$$a = \arg\max_a (Q(s,a) + U(s,a)) \tag{2.17}$$

The $U(s,a)$ term is an exploration bonus, which depends on the prior $P(s,a)$

and visit $N(s,a)$ statistics of the edge, and is used to ensure that the search does not select actions greedily under the current value estimates. Many different flavours of the UCT formula have been proposed in the literature to solve the above exploration-exploitation problem [6, 7]. A particularly interesting approach for this work, is pUCT which was used in the *AlphaGo* agent[108]. pUCT (eq 2.18) biases the search to explore edges which seem most promising according to their prior $P(s,a)$.

$$U(s,a) = c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \tag{2.18}$$

The c_{puct} term is a constant used to control the level of exploration, while the term $\sum_b N(s,b)$ is equal to the times the node s was visited.

Expansion The selection phase is terminated when a leaf node s_{leaf} has been reached and an edge (s_{leaf}, a) has been selected. Subsequently, the expansion phase takes place, where a new node is added to the tree and all its edges are initialized.

Rollout In the rollout phase, a *rollout-policy* is used to compute an estimate of the value function of the new node s_{leaf}. The estimate is obtained by executing the rollout-policy starting at the state s_{leaf} until the end of the episode and computing the empirical return. We can increase the accuracy of the estimate and reduce its variance by executing multiple such rollouts at the expense of a higher computational cost.

Backup The final phase of a simulation is the backup. During this phase the statistics of each traversed edge are updated using the value estimate of the newly added node s_{leaf}.

$$\begin{aligned} Q(s,a) &:= \frac{\hat{Q}(s,a) + N(s,a)Q(s,a)}{1+N(s,a)} \\ N(s,a) &:= N(s,a) + 1 \end{aligned} \tag{2.19}$$

The term $\hat{Q}(s,a)$ corresponds to the Q estimate of the edge given the value of the leaf V_{leaf}. In two-player zero sum games where a discount of 1 is usually assumed and there is only a terminal win-loss reward this is computed as follows:

$$\hat{Q}(s,a) = \begin{cases} V_{leaf}, & \text{if } Player(s) = Player(s_{leaf}) \\ -V_{leaf}, & \text{otherwise} \end{cases} \tag{2.20}$$

In the general case of an MDP with rewards $r \in R$ and discount γ, the $\hat{Q}$ is computed using the n-step returns:

$$\hat{Q}(s,a) = \sum_{\tau=0}^{l} \gamma^{\tau} r_{\tau} + \gamma^{l} V_{leaf} \tag{2.21}$$

Where $r_{\tau}, \tau = 0, \cdots l$ are the intermediate rewards observed while transitioning from state s to the leaf state s_{leaf} and l is the length of this transition.

The MCTS algorithm terminates after a pre-specified number of simulations have completed. Figure 2.1 illustrates the 4 phases of a MCTS simulation.

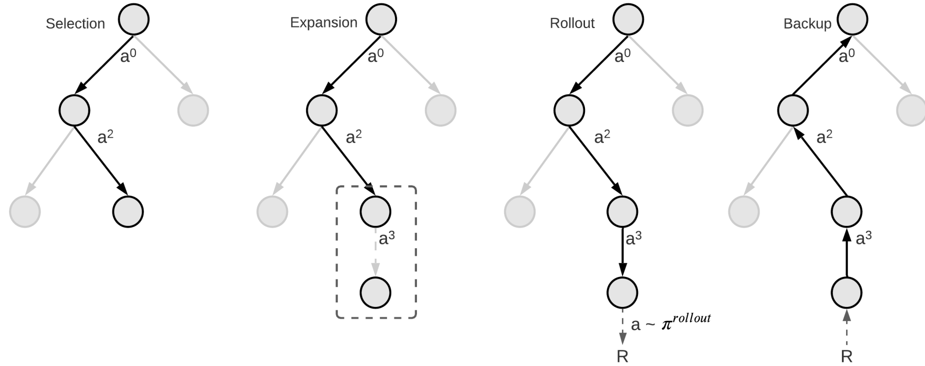


Figure 2.1: **A Monte-Carlo Tree Search simulation**. Each simulation is comprised of four phases: *selection*, *expansion*, *rollout* and *backup*. During the selection phase the tree is traversed starting from the root node until a leaf edge is reached. The edges inside the tree are selected by computing the pUCT [99] formula. In the expansion phase a new node along with all its edges is added to the tree. A value estimate for the newly added node is computed during the rollout phase, by running games of selfplay starting at the current position until the end of the game and selecting actions using a rollout policy $\pi^{rollout}$. Finally, the statistics of the affected sub-tree are adjusted based on the value of the new node.

At the end of the search, the algorithm selects the action at the root node which had the most visits, as its best estimate for the optimal action. MCTS is an any-time algorithm, meaning that at any time the user has access to its current best estimate of the optimal action, and it provably converges to the optimal solution [70].

Chapter 3

Games for Reinforcement Learning

3.1 Board games

Board games have been extensively studied in the field of artificial intelligence [134, 10, 131, 72, 122, 19, 106, 20, 57]. Games such as chess or Go have been heavily used as testbeds for the development of new ideas and have significantly boosted the progress of the field. There are a number of benefits with the use of such games:

- In contrast to the real world, they are constrained and well defined.
- They are challenging to humans.
- They provide an easy way to measure progress by comparing playing strengths between methods.
- There is an extended literature of game analysis and past games available.

In this section, we provide a short description of the board games used in this work, along with a review of notable previous work.

3.1.1 Go

The game of Go is a classic oriental two player board game. The game is played on a 19x19 grid, however, smaller boards of sizes 13x13 or 9x9 are also commonly used. The players use colored stones, black for the first and white for the second, to make their moves on the board. Each player can place their stone on an empty intersection. The stones cannot be moved after they have been placed, unless they

are captured by the opponent player. Adjacent stones of the same color are called blocks and they define the territory of each player. The number of empty intersections adjacent to a block are called its liberties. A block is captured and taken off the board when it has zero liberties. The goal of each player is to maximize their territory on the board. A game ends when both players pass and the winner is the player with the greatest territory. To account for the advantage of the first (black) player, the second player (white) receives an extra bonus, known as komi, which is added to its final territory.

Numerous reinforcement learning approaches have been applied to the game of Go. *RLGO* [111] combines planning with a learned value function. It uses a hybrid regime which integrates a variant of Dyna [112] with an alpha-beta search. The value function is represented as a linear combination of all possible 3x3 stone patterns and it was trained via selfplay using temporal difference learning [117] to predict the final outcome of the game. The final agent achieved weak amateur strength in 9x9 Go. *NeuroGo* [34, 33] utilized a neural network to learn a local value function for each intersection of the board. The network was trained via temporal-difference learning to predict the probability that a given intersection will be part of the black player's territory at the end of the game. *NeuroGo* made extensive use of Go specific knowledge which was incorporated in the design of the network architecture.

The most successful computer Go programs utilize a MCTS search during play. Prior to *AlphaGo*, programs such as CrazyStone [27], Pachi [9] and Fuego [35] combined a MCTS search with a fast rollout policy which was based on handcrafted features and made use of substantial domain specific expertise, achieving strong amateur level performance in 19x19 Go.

3.1.2 Chess

The game of chess is the most widely-studied domain in the history of artificial intelligence and computer chess is as old as computer science itself. Charles Babbage, Alan Turing, Claude Shannon, and John von Neumann devised hardware, algorithms and theory to analyse and play the game of chess. Chess subsequently

became a grand challenge task for a generation of artificial intelligence researchers, culminating in high-performance computer chess programs that play at a super-human level [20, 57].

There is an extensive literature on the application of reinforcement learning techniques to the game of chess. *GnuChess* [40] is an open source chess engine, used for research purposes, which combines a principal variation search with a heuristic evaluation function. *NeuroChess* [127] evaluates positions by a neural network that uses 175 handcrafted input features. It was trained by temporal-difference learning to predict the final game outcome, and also the expected features after two moves. *NeuroChess* won 13% of games against *GnuChess* using a fixed depth 2 search. Beal and Smith applied temporal-difference learning to estimate the piece values in chess [11], starting from random values and learning solely by self-play. *KnightCap* [10] evaluates positions by a neural network that uses as input frames an attack table based on knowledge of which squares are attacked or defended by which pieces. It was trained by a variant of temporal-difference learning, known as TD(leaf), that updates the leaf value of the principal variation of an alpha-beta search. *KnightCap* achieved human master level after training online against strong opponents with hand-initialized piece-value weights. *Meep* [131] evaluates positions by a linear evaluation function based on handcrafted features. It was trained by another variant of temporal-difference learning, known as TreeStrap, that updates all nodes of an alpha-beta search. *Meep* defeated human international master players in 13 out of 15 games, after training by self-play with randomly initialized weights. *Giraffe* [72] evaluates positions by a neural network that includes mobility maps and attack and defend maps describing the lowest valued attacker and defender of each square. It was trained by self-play using TD(leaf), also reaching a standard of play comparable to international masters. *DeepChess* [32] trained a neural network to perform pair-wise evaluations of positions. It was trained by supervised learning from a database of human expert games that was pre-filtered to avoid capture moves and drawn games. *DeepChess* reached a strong grandmaster level of play.

3.1.3 Shogi

Shogi is a classic two player board game, played on a 9x9 board. It is a Japanese variant of chess, with the fundamental difference that captured pieces can re-enter the board and be used by the opponent. There are 8 different types of pieces in Shogi: pawns (9), lances(2), knights(2), silver generals(2), gold generals(2), bishop(1), rook(1) and king(1). Each player places their pieces to face their opponent and their rank is visible to both players. Similarly to chess the objective of the game is to achieve a checkmate. Each piece type moves differently: a king can move one space in any direction, a gold general can move forward, diagonally forward, sideways or backwards, a silver general can move forward, diagonally forward, or diagonally backwards, a knight can move over other pieces by taking two steps forward and then one sideways, a lance can take any number of forward steps, rooks move orthogonally, a bishop can move diagonally, and finally pawns can only move one space forward. At any point during play two pieces cannot occupy the same square. An opponent piece can be captured and be taken off the board by moving a player's own piece to its square. Captured pieces can re-enter the board and be used by the player as its own. Pieces can also be promoted by moving them within the last three rows of the board from the player's perspective (promotion zone).

Similarly, to chess, Shogi has been used as a testbed for the development of new methods in artificial intelligence. Beal and Smith [12] applied their method to Shogi learning solely by self-play. Kaneko and Hoki [67] trained the weights of a shogi evaluation function comprising a million features, by learning to select expert human moves during alpha-beta search. They also performed a large-scale optimization based on minimax search regulated by expert game logs [54]; this formed part of the *Bonanza* engine that won the 2013 World Computer Shogi Championship.

3.1.4 Backgammon

Backgammon is an ancient two player, zero-sum, perfect information, stochastic board game. The board consists of 24 squares (or points) and each player controls 15 checkers, which can move based on the outcome of a dice roll. The two play-

ers move their checkers in opposite directions and their goal is to move all their checkers off the board first. In addition to a simple win, a player can also score a double (*gammon*) or a triple (*backgammon*) win. A *gammon* is achieved when a player bears off all their checkers before their opponent manages to bear off any, while a *backgammon* when the opponent also has checkers left in the player's home quadrant (farthermost quadrant from the opponent's perspective). Each player can impede the progress of their opponent through *hitting* the opponent's checkers or blocking their advancement. A *hit* is achieved when a player's checker advances to a position with a single opponent's checker. Then the opponent's checker needs to reenter the board in the player's home quadrant and no further moves are allowed to the opponent until that happens. A position is blocked to the opponent when it is occupied by at least two of the player's checkers. Each player makes moves based on the values yielded by rolling two dice. In the case of *doubles*, aka the two dice have the same value, the player can play up to 4 moves.

One of the challenges of computer Backgammon is its high branching ratio, since at each ply there are 21 chance outcomes, which yield positions with an average of 20 legal moves each, resulting in a branching ratio of several hundred per ply. In the field of artificial intelligence, backgammon was popularized as a standard testbed by *TD-gammon* [122]. *TD Gammon* evaluated positions by a multilayer perceptron, trained by temporal-difference learning to predict the final game outcome. When its evaluation function was combined with a 3-ply search [124] TD Gammon surpassed the playing ability of world-champion human players. A subsequent paper introduced the first version of Monte-Carlo search [125], which evaluated root positions by the average outcome of n-step simulations. Each simulation was generated by greedy move selection and the nth position was evaluated by TD Gammon's neural network.

3.2 Video games

The recent developments in the field of deep neural networks and reinforcement learning has led to the adoption of video games as the ideal testbeds for the devel-

opment and evaluation of new algorithms. Video games provide a wide range of challenging tasks, a well defined reward signal in the form of the game score, and allow for fast experimentation given their modest computational cost. In this thesis, we used the Atari suite [15] and the *2048* games to assess the performance of our methods.

3.2.1 Atari

The Atari suite of environments [15] has been heavily used in the literature for evaluating deep reinforcement learning algorithms. Here, we enumerate some notable approaches that led to significant improvements.

One of the first algorithms which successfully combined deep neural network with reinforcement learning was *DQN* [84]. *DQN* utilizes a deep convolutional network to represent a Q function, which is trained via Q-learning. One of the key difficulties in combining deep network function approximators with RL losses, is the non-stationarity of the data distribution induced by the RL loop. *DQN* overcomes this problem by employing a transition table which stores a history of past state-action transitions, and by using a separate network to compute the Q-function targets. This separate network, called target network, is updated every k training steps to match the online Q network.

In later work, the performance of deep reinforcement learning algorithm was greatly benefited by new improvements in three main areas: massive parallelization of the RL loop, improvements in the architecture of the transition table and finally better learning rules.

The *Gorila* [85] framework managed to significantly improve the performance of the *DQN* algorithm by massively parallelising both the generation of experience by the agent and the network updates. In their approach multiple actors generated experience by interacting with separate instances of the environment, while multiple learners synchronously updated the network. By using multiple actors, their method reduced the correlation between the training samples and improved the learning efficiency of the algorithm. At the same time the use of multiple learners allowed for the use of larger batch sizes, thus reducing the variance of the gradient updates.

In [101] a new variant of the transition table was introduced which prioritized different state-action transitions based on their observed TD error. By increasing the sampling frequency of transitions with a high TD error, the learning was biased towards transitions which the Q-network found surprising or novel. This approach was further extended in [55], where it was applied in a massively parallel setup.

A new highly parallelizable asynchronous actor-critic approach was proposed in [83] (see 2.4). In this setup, multiple actor-learner pairs asynchronously updated a shared common network. This method managed to significantly outperform a *DQN* baseline without requiring access to any specialized hardware (GPU). A synchronous variant of this method was proposed in [36]. There, the learner updates were synchronized and efficiently computed on a GPU.

Extending value functions to represent distributions over discounted returns instead of expectations has been another transformative idea that has led to significant increases in performance [14, 29, 28]. In this approach, the TD-error is replaced by a distributional equivalent which measures the distance of the current prediction and the n-step target in the space of distributions. By considering full distributions instead of expectations, distributional value functions can better account for the uncertainty in the value estimates, which can be beneficial in terms of learning efficiency and performance.

3.2.2 2048

The game of 2048 is a single player, perfect information, stochastic puzzle game. The board is represented by a 4x4 grid with numbered tiles, and at each step the player has four possible actions which correspond to the four arrow keys (up, down, right, left). When the player selects an action, all the tiles in the board slide in the corresponding direction until they reach the end of the board or another tile of different value. Tiles of the same value are merged together to form a new tile with a value equal to their sum, and the resulting value is added to the running score of the game. After each move, a new tile randomly appears in an empty spot on the board with a value of 2 or 4. The game ends when there are no more moves available to the player that can alter the board state.

There is a plethora of previous work [120, 136, 88, 97, 86] on combining reinforcement learning and tree search methods for tackling *2048*. Despite its simplicity, model-free approaches have traditionally struggled to achieve high performance, while planning-based approaches have exploited perfect knowledge of the simulator. To date, the best performing agent uses the planning-based approach proposed in [63]. This method uses an expecti-max tree search over a perfect simulator, combined with domain-specific knowledge and a number of novel algorithmic ideas that exploited the structure of this specific problem.

Chapter 4

AlphaGo

In this chapter we describe the *AlphaGo* algorithm, which has acted as the main source of inspiration for the work presented in this thesis.

4.1 Introduction

The game of Go has been traditionally viewed as a grand challenge in the field of artificial intelligence [81]. Go is a perfect information zero-sum game, meaning that all the information about the board state is available to both players and the rewards for the two players sum to zero $r_1 = -r_2$. In that respect, Go is similar to other classic board games, like chess, which have been studied heavily in the field of AI (see section 3.1.2). However, Go poses new challenges given its huge search space and the difficulty of evaluating board positions and moves. While previous approaches that combined an Alpha-Beta search with a heuristic evaluation function achieved super-human performance in chess and checkers, they only achieved weak amateur level playing strength in Go.

AlphaGo[108] was the first computer program to achieve superhuman performance in Go. It defeated the European champion Fan Hui in October 2015. In March 2016, it defeated Lee Sedol, the winner of 18 world championship titles, by a score of 4-1. *AlphaGo* combines deep neural networks for evaluating board positions and moves with a Monte Carlo Tree Search (see section 2.6.2) for planning. It uses two networks, a *policy* which given a board position produces a list of promising moves, and a *value* network which evaluates a board and estimates

the winning probability from that position for each player. The neural networks are trained separately by utilizing expert human data and reinforcement learning techniques. During play, starting from the current board state, MCTS uses the two networks to prune the search space and explore the most promising game variations.

4.2 Algorithm

In this section we describe the two main components of the *AlphaGo* algorithm, namely the neural networks and the MCTS search.

4.2.1 Networks

Policy The policy network, shown in figure 4.1 **A**, is employed by *AlphaGo* to produce a list of promising actions for the board positions encountered during its search. The network receives a state s as an input and it produces a probability distribution over actions a, $p_\sigma(a \mid s)$. This network was trained using supervised learning to predict expert moves. Specifically, the policy weights were updated using stochastic gradient ascent to maximize the likelihood of the expert action a given the board state s:

$$\Delta\sigma \propto \nabla_\sigma p_\sigma(a \mid s) \tag{4.1}$$

The training dataset was generated using human games stored in the KGS Go server [1], and it was comprised of 30 million positions. The policy was represented by a 13 layers deep convolutional neural network with weights σ and rectifier non-linearities. The distribution over the legal actions was produced by a final softmax layer.

Along with the main policy network, *AlphaGo* employs a second smaller policy network comprised of a single linear layer, during its search. This network is called during the rollout phase of the MCTS algorithm (see 2.6.2).

Value *AlphaGo* uses a value network, shown in figure 4.1 **B**, to evaluate the board positions encountered during its search. The value network receives a board state s as an input and produces a scalar value $\hat{V}^\pi(s)$ which predicts the expected final outcome of the game when both players select actions using policy π. Ideally, the acting policy would correspond to the optimal policy π^* and the value network

would return an estimate of the optimal value function. However, since π^* cannot be obtained, *AlphaGo* uses a different strong policy π_ρ trained via selfplay reinforcement learning to generate training targets for the value network. This policy was again represented by a deep neural network and it was initialized to be the same as the policy network obtained via supervised learning (see 4.2.1). Subsequently, its playing strength was improved via selfplay using a variant of the REINFORCE learning algorithm (see 2.4). This improved policy was used to generate a training dataset of state outcome pairs (s,z). The value network was trained on this dataset by regression using stochastic gradient descent to minimize the mean squared error (MSE) between the network output $V(s)$ and the corresponding outcome z. Finally, it used a similar architecture and input representation as the policy network.

4.2.2 Tree Search

The key idea of *AlphaGo* is to combine the policy and value networks with a MCTS planning algorithm. To accommodate this, *AlphaGo* adapts MCTS to efficiently use the two networks in the following ways:

- When a new node is added, during the expansion phase, the two networks receive the corresponding board position and compute the policy and value for this state.
- During the selection phase the output of the policy network is used in the pUCT formula (see 2.6.2).
- In the simulation phase the board position is evaluated by combining the output of the value network, with the Monte Carlo return computed using its rollout policy network.

The use of neural networks inside the search increases the computational demands of the classic MCTS search. To address this, *AlphaGo* uses a multi-threaded implementation of MCTS with asynchronous evaluations and utilizes GPUs for computing the policy and value networks in parallel. At each point several threads traverse the tree from the root node until they reach a leaf edge. When a thread reaches

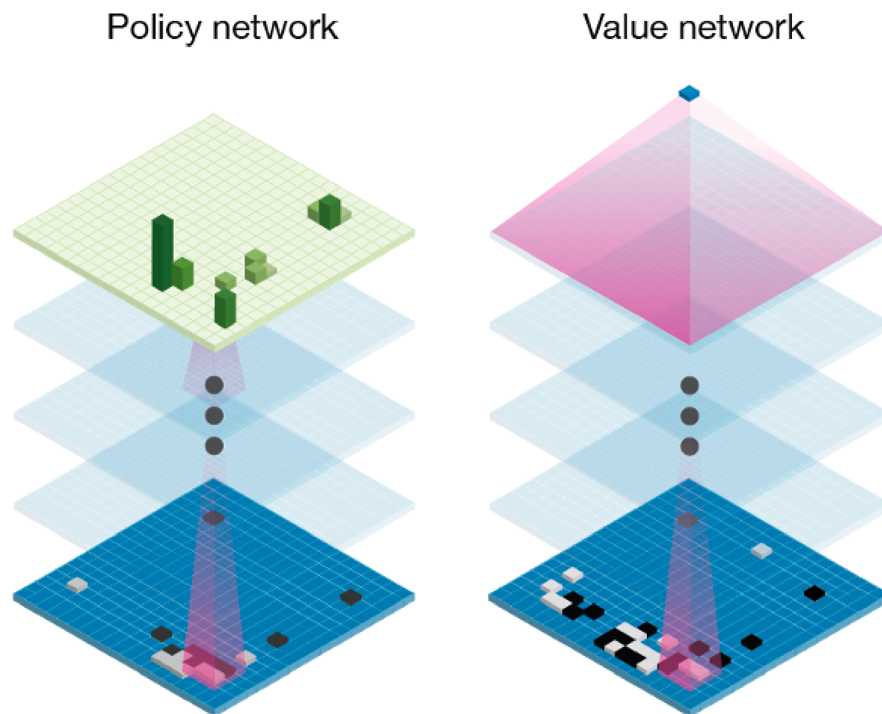


Figure 4.1: **The policy and value networks of *AlphaGo*** **A** The policy network of *AlphaGo* receives a board position s as input and generates a distribution over possible actions. **B** The value network of *AlphaGo* evaluates a board position s and returns the expected game outcome for the current player.

a leaf edge it sends an expansion request to a separate network evaluator which batches multiple requests together and computes their output on the GPU. This way *AlphaGo* can increase the size of its search and simultaneously increase its playing strength for a given time limit by employing more CPUs and GPUs. The published version of the algorithm used 40 search threads, 48 CPUs and 8 GPUs, while the authors also showed that their approach can scale to use multiple machines with 1202 CPUs and 176 GPUs.

4.3 Limitations

Despite its success *AlphaGo* suffers from a number of limitations which hinders its applicability to problems other than Go.

Human data *AlphaGo* uses a significant amount of human expert data to train its policy network. In many interesting real world problems such data are limited or not available. Furthermore, by utilizing human data, we introduce strong human biases in the solutions considered and produced by the algorithm, which can hinder its final performance [49].

Domain specific knowledge *AlphaGo* makes use of many domain specific heuristics and hand-crafted features. The board representation which is processed by *AlphaGo*'s networks includes Go specific statistics which were found to increase the final accuracy of the policy network, while the simulation-policy uses extra hand-crafted features which led to a stronger playing strength. Moreover, *AlphaGo* exploited the inherent symmetries found in the game of Go during training for data augmentation. Devising those heuristics involved years of research and required expertise in the game of Go.

Computational requirements and complexity Arguably, *AlphaGo* has high computational requirements both during training and deployment. It requires the deployment of different training pipelines which use both supervised and reinforcement learning techniques. Obtaining the final networks involved training a policy network, then improving its strength via selfplay RL, subsequently, generating a dataset for the value network and finally learning a value function. During deployment *AlphaGo* uses hundreds of CPUs and GPUs. As a result, applying *AlphaGo* to a new problem requires both the implementation of multiple pipelines and the use of a significant amount of computational resources.

Part II

Tree Search Planning with Deep Networks

Chapter 5

AlphaZero

A long-standing goal of artificial intelligence is an algorithm that learns, tabula rasa, superhuman proficiency in challenging domains. As we saw, *AlphaGo* became the first program to defeat a world champion in the game of Go. The tree search in *AlphaGo* evaluated positions and selected moves using deep neural networks. These neural networks were trained by supervised learning from human expert moves, and by reinforcement learning via self-play. In this section we introduce a new algorithm, *AlphaZero*[1], which is based solely on reinforcement learning, without human data, guidance, or domain knowledge beyond game rules, and which achieves superhuman performance in many challenging domains. Similarly to *AlphaGo*, *AlphaZero* combines deep neural networks with a MCTS search, but, unlike *AlphaGo*, it trains its policy and value networks completely from scratch via reinforcement learning from games of selfplay. The policy is trained to predict the move selection generated by the search and the value is trained to predict the final outcome in games of self-play. We applied our new method to the highly complex domains of Go, chess and shogi. Within 24 hours, starting from random play, *AlphaZero* achieved a superhuman level of play and convincingly defeated a world champion program in each case.

[1] The *AlphaZero* algorithm was previously published in [110, 109]

5.1 Introduction

Artificial intelligence aims to create systems that can match or exceed human performance in a wide range of challenging domains. A natural step in this direction, which has been studied heavily in the literature, is to design algorithms that learn to imitate the decisions made by human experts [48, 74, 71, 50]. However, this dependence on expert data prohibits the application of these methods in many real-world problems where such data are expensive, unreliable, or simply unavailable. A different approach is to develop domain specific systems which make use of heuristics and hand-crafted features designed by human experts. Such systems have been successfully applied to complex domains like chess, exemplified by the defeat of the human world champion by Deep Blue in 1997 [20]. Nonetheless, these domain specific solutions fail to generalize to other problems, require a substantial human effort to be developed and by incorporating significant amounts of human prior knowledge their final performance is limited by the choices of their designers [49]. Reinforcement learning systems can overcome these limitations by learning from their own experience through trial and error [100, 122]. This allows them, at least in principle, to exceed human capabilities and to operate in domains where human expertise is lacking.

AlphaZero is a general reinforcement learning algorithm inspired by *AlphaGo*, the first program to achieve superhuman performance in Go. *AlphaGo* demonstrated that the playing strength of a policy and value network can be significantly improved by combining them with a tree search. The main idea of *AlphaZero* is to exploit this result in a reinforcement learning setting, by utilizing the tree search as a powerful policy improvement operator. Starting from randomly initialized weights, *AlphaZero* generates games of selfplay by running a tree search at each step, and then uses the improved search policy and game outcomes as training targets for the policy and value respectively. Unlike *AlphaGo*, the policy and value function are represented by a single network with two output heads, which improve the representation learning efficiency of the algorithm. The network input representation simply encodes the board observation of each domain and does not include

any hand-crafted features. Finally, the tree-search implemented by *AlphaZero* relies only upon its network predictions to evaluate positions and sample moves, and it does not perform any Monte Carlo simulations.

We applied *AlphaZero* to the games of Go, chess and shogi, without any additional domain knowledge, except the rules of the game, demonstrating that a general-purpose reinforcement learning algorithm can achieve, tabula rasa, superhuman performance across many challenging domains. In each environment we evaluated our method against the best performing computer program in a series of matches. In Go, we compared *AlphaZero* against *AlphaGo*, where we considered two different version of the algorithm: *AlphaGo Fan* which defeated the European champion Fan Hui in October 2015, and *AlphaGo Lee* a subsequent version which defeated Lee Sedol, the winner of 18 international titles, in March 2016. For chess, we used *Stockfish*, the world-champion in the 2016 Top Chess Engine Championship (TCEC), to assess the playing strength of *AlphaZero*. Finally in Shogi, *AlphaZero*'s playing strength was evaluated against *Elmo*, the Computer Shogi Association world-champion which has previously defeated human champions [5].

5.2 Algorithm

5.2.1 Network

AlphaZero uses a deep neural network f_θ with parameters θ, which at each time step t takes a history of raw board observations $s_t = \{o_1, ..., o_t\}$ as an input and outputs both move probabilities and a value, $\mathbf{p}, v = f_\theta(s_t)$. The vector of move probabilities $\mathbf{p}$ represents the probability of selecting each move, $p_a = Pr(a \mid s_t)$. v is the value estimate for this position s_t produced by the network. Intuitively, in board games, the value estimates the expected game outcome or in other words, the probability of the current player winning from position s_t. This neural network combines the roles of both policy network and value network of *AlphaGo* into a single architecture.

5.2.2 Search

AlphaZero combines its network with a MCTS search (see 2.6.2). The search uses the neural network f_θ to guide its simulations, as shown in Figure 5.1. During

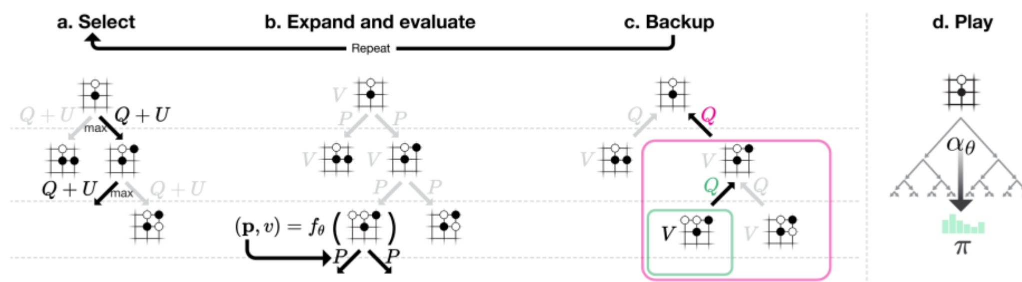


Figure 5.1: **Monte-Carlo tree search in *AlphaZero*** **a** During the selection phase, starting from the root node, the tree is traversed by selecting edges using the pUCT formula [99] (see 2.6.2) until a leaf node is reached. The pUCT formula combines the current value estimate for the edge Q with an exploration bonus term U which depends on the stored prior probability P and the visit count N of the edge. **b** At the expansion phase, a new node is added to the tree and the associated position s is evaluated by the neural network $(P(s,\cdot),V(s)) = f_\theta(s)$. **c** At the end of each simulation, the value estimates of the tree edges are updated to track the mean of all evaluations V in their corresponding subtree. **d** Once the search is complete, search probabilities π are returned, proportional to $N^{1/\tau}$, where N is the visit count of each move from the root state and τ is a parameter controlling temperature.

the selection phase the move probabilities produced by the network are used in the pUCT formula to bias the search towards the most promising moves. When a new node is expanded the corresponding state s is evaluated by the network f_θ, and its outputs are used to initialize the node's value estimate and its edges. *AlphaZero*'s search does not use Monte-Carlo rollouts and depends only on the output of the network to evaluate positions.

5.2.3 Self-play

AlphaZero generates training data for its network from games of self-play as shown in figure 5.2 **a**. At each position s a MCTS search is executed using the latest network f_θ and actions are selected based on the policy produced by it. This search-policy π^{MCTS} is obtained by considering the visit counts of the edges of the root node, $\pi^{MCTS}(a \mid s) \propto N(s,a)$. Intuitively, since MCTS tends to explore more edges

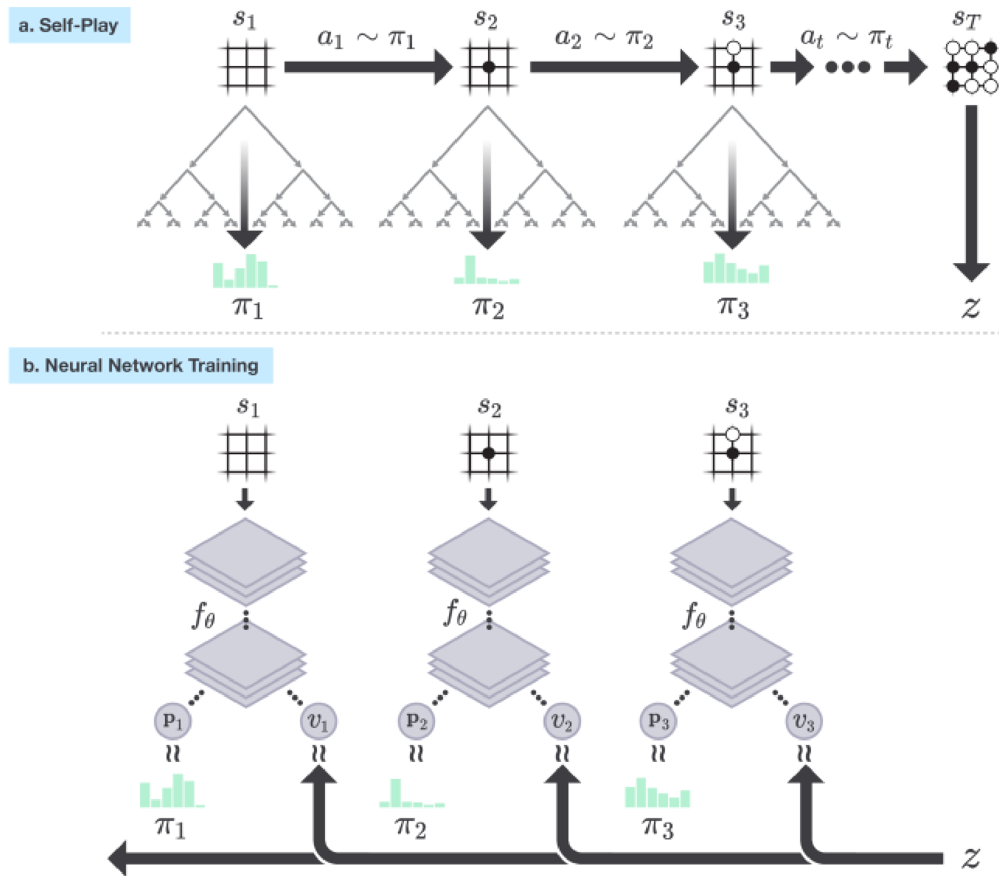


Figure 5.2: **Self-play reinforcement learning in *AlphaZero* a** *AlphaZero* generates a game of selfplay by executing a MCTS search at each step using the latest neural network f_θ. The actions are selected based on the search probabilities produced by the search. **b** *AlphaZero*'s neural network takes the current board position s as an input and outputs a vector $\mathbf{p}$ of move probabilities and a scalar value v which represents the probability of the current player winning the game starting at position s. The neural network parameters θ are updated so as to maximise the similarity of the policy vector $\mathbf{p}$ to the search probabilities π^{MCTS} , and to minimise the error between the value prediction v and the game outcome. The new parameters are used in the next iteration of self-play a.

which lead to better play, the visit count is a measure of how promising the search thinks each edge is. As is common practice, the resulting policy can be adjusted using a temperature parameter τ $\pi^{MCTS}(a \mid s) \propto N(s,a)^{1/\tau}$, where we use the convention that $\tau = 0$ corresponds to a greedy policy.

For each episode of selfplay the algorithm stores the search-policy and the outcome of the game. These statistics are then used as targets for the policy and value network. By running a search at each time-step, *AlphaZero* ensures that its search policy is much stronger than the raw network policy it used inside its search. At the same time, this improved policy generates games of stronger play which are used to train a better evaluation function. This process of improving the quality of the network using search and subsequently using the improved network to obtain an even stronger search policy is iterated multiple times in *AlphaZero*.

5.2.4 Training

The network uses the experience generated during the previous iterations of self-play to update its weights θ. At each training step i it samples a batch of tuples (s, π^{MCTS}, z), where s is the board position, π^{MCTS} the move probabilities generated by MCTS at this position, and z the outcome of the game from the perspective of the current player ($+1$, -1 or 0 if the player won, lost or draw respectively). Subsequently, the weights are updated using gradient descent to minimize the following loss:

$$\mathbf{p}, v = f_\theta(s) \qquad l = (z - v)^2 - \pi^{MCTS} log \mathbf{p} + c\|\theta\|^2 \tag{5.1}$$

The value v is trained to predict the self-play winner z and move probabilities $\mathbf{p}$ are adjusted to match the moves distribution produced by the MCTS search. An extra L2 regularization loss $\|\theta\|^2$ controlled by a parameter c is added to the loss to combat overfitting.

5.3 Experiments

To evaluate the performance of our *AlphaZero* agent we applied it to the games of Go, chess and shogi. For our initial experiments we focused on the game of Go, where we made used of the symmetries present in the game during both training

and selfplay. This version, called *AlphaGo Zero*, preceded our final version which was invariantly applied to all three board games. In this section, we present our experiments starting with the early *AlphaGo Zero* work followed by the final results obtained using the general *AlphaZero* algorithm.

5.3.1 *AlphaGo Zero*

AlphaGo Zero exploits the fact that the rules of Go are invariant under rotation and reflection; this knowledge is utilised both by augmenting the data set during training to include rotations and reflections of each position, and to sample random rotations or reflections of the position during MCTS. Moreover, it generates self-play games using the best player from all previous iterations. After each iteration of training, the performance of the new player is measured against the best player; if the new player wins by a margin of 55% then it replace the best player. Those domain specific adaptations of the *AlphaZero* algorithm were removed in our later experiments (see 5.3.2).

Results

AlphaGo Zero was trained, starting from completely randomly initialized weights, for approximately 3 days. It generated 4.9 million games of selfplay using a search budget of 1,600 simulations at each time step. It was trained for a total of 700,000 steps using a mini-batch size of 2,048 positions. The neural network was represented by 20 residual blocks [50] of convolutional layers [75, 42] with batch normalization [60] and rectifier non-linearities (see A.2.2 for a detailed description). Figure 5.3 shows the performance of *AlphaGo Zero* during self-play reinforcement learning, as a function of training time, on an Elo scale [27].

Surprisingly, *AlphaGo Zero* outperformed *AlphaGo Lee* after just 36 hours; for comparison, *AlphaGo Lee* was trained over several months. After 72 hours, we evaluated *AlphaGo Zero* against the exact version of *AlphaGo Lee* that defeated Lee Sedol, under the 2 hour time controls and match conditions as were used in the man-machine match in Seoul. *AlphaGo Zero* used a single machine with 4 Tensor Processing Units (TPUs)[65], while *AlphaGo Lee* was distributed over many

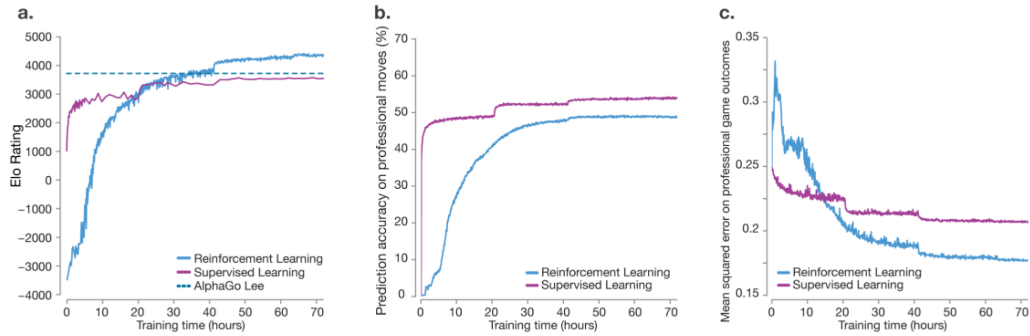


Figure 5.3: **Empirical evaluation of *AlphaGo Zero*** **a** Performance of self-play reinforcement learning. The plot shows the performance of each MCTS player π^{MCTS} from each iteration i of reinforcement learning in *AlphaGo Zero*. Elo ratings were computed from evaluation games between different players, using 0.4 seconds of thinking time per move. For comparison, a similar player trained by supervised learning from human data, using the KGS data-set, is also shown. **b** Prediction accuracy on human professional moves. The plot shows the accuracy of the neural network f_θ, at each iteration of self-play i, in predicting human professional moves from the GoKifu data-set. The accuracy measures the percentage of positions in which the neural network assigns the highest probability to the human move. The accuracy of a neural network trained by supervised learning is also shown. **c** Mean-squared error (MSE) on human professional game outcomes. The plot shows the MSE of the neural network f_θ, at each iteration of self-play i, in predicting the outcome of human professional games from the GoKifu data-set. The MSE is between the actual outcome $z \in \{-1,+1\}$ and the neural network value v, scaled by a factor of $1/4$ to the range $[0,1]$. The MSE of a neural network trained by supervised learning is also shown.

machines and used 48 TPUs. *AlphaGo Zero* defeated *AlphaGo Lee* by 100 games to 0.

We subsequently applied our reinforcement learning pipeline to a second instance of *AlphaGo Zero* using a larger neural network and over a longer duration. Training again started from completely random behaviour and continued for approximately 40 days. Over the course of training, 29 million games of self-play were generated. Parameters were updated from 3.1 million mini-batches of 2,048 positions each. The neural network contained 40 residual blocks. The learning curve is shown in figure 5.4 **a**.

Finally, we evaluated the fully trained *AlphaGo Zero* using an internal tournament against *AlphaGo Fan*, *AlphaGo Lee*, and several previous Go programs. We also played games against the strongest existing program, *AlphaGo Master* – a program based on the algorithm and architecture presented in this work but utilising human data and features – which defeated the strongest human professional players 60–0 in online games in January 2017 [58]. In our evaluation, all programs were allowed 5 seconds of thinking time per move; *AlphaGo Zero* and *AlphaGo Master* each played on a single machine with 4 TPUs; *AlphaGo Fan* and *AlphaGo Lee* were distributed over 176 GPUs and 48 TPUs respectively. We also included a player based solely on the raw neural network of *AlphaGo Zero*; this player simply selected the move with maximum probability. Figure 5.4 **b** shows the performance of each program on an elo scale [27]. The raw neural network, without using any lookahead, achieved an Elo rating of 3,055. *AlphaGo Zero* achieved a rating of 5,185, compared to 4,858 for *AlphaGo Master*, 3,739 for *AlphaGo Lee* and 3,144 for *AlphaGo Fan*. *AlphaGo Zero* was also evaluated in a series of a 100 head to head matches against *AlphaGo Master* with 2 hour time controls. *AlphaGo Zero* won by 89 games to 11.

Ablations

To assess the merits of self-play reinforcement learning, compared to learning from human data, we trained a second neural network (using the same architecture) to predict expert moves in the KGS data-set; this achieved state-of-the-art prediction

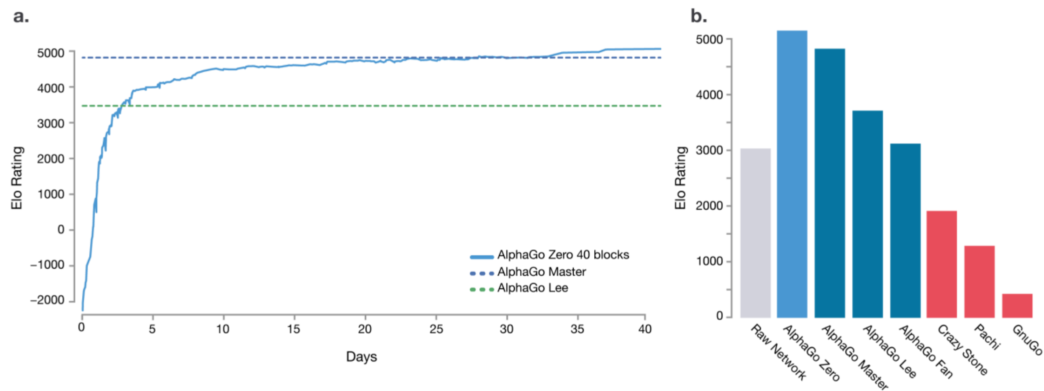


Figure 5.4: **Performance of *AlphaGo Zero*.** **a** Learning curve for *AlphaGo Zero* using a larger 40 block residual network over 40 days. The plot shows the performance of each player π^{MCTS} from each iteration i of our reinforcement learning algorithm. Elo ratings were computed from evaluation games between different players, using 0.4 seconds per search. **b** Final performance of *AlphaGo Zero*. *AlphaGo Zero* was trained for 40 days using a 40 residual block neural network. The plot shows the results of a tournament between: *AlphaGo Zero*, *AlphaGo Master* (defeated top human professionals 60-0 in online games), *AlphaGo Lee* (defeated Lee Sedol), *AlphaGo Lee* (defeated Fan Hui), as well as previous Go programs Crazy Stone, Pachi and GnuGo. Each program was given 5 seconds of thinking time per move. *AlphaGo Zero* and *AlphaGo Master* played on a single machine on the Google Cloud; *AlphaGo Fan* and *AlphaGo Lee* were distributed over many machines. The raw neural network from *AlphaGo Zero* is also included, which directly selects the move a with maximum probability, without using MCTS. Programs were evaluated on an Elo scale [27]: a 200 point gap corresponds to a 75% probability of winning.

	KGS train	KGS test	GoKifu validation
Supervised learning (20 block)	62.0	60.4	54.3
Supervised learning (12 layer)	59.1	55.9	-
Reinforcement learning (20 block)	-	-	49.0
Reinforcement learning (40 block)	-	-	51.3

Table 5.1: **Move prediction accuracy.** Percentage accuracies of move prediction for neural networks trained by reinforcement learning (i.e. *AlphaGo Zero*) or supervised learning respectively. For supervised learning, the network was trained for 3 days on KGS data (amateur games); comparative results are also shown from Silver[108]. For reinforcement learning, the 20 block network was trained for 3 days and the 40 block network was trained for 40 days. Networks were also evaluated on a validation set based on professional games from the GoKifu data set.

	KGS train	KGS test	GoKifu validation
Supervised learning (20 block)	0.177	0.185	0.207
Supervised learning (12 layer)	0.19	0.37	-
Reinforcement learning (20 block)	-	-	0.177
Reinforcement learning (40 block)	-	-	0.180

Table 5.2: **Game outcome prediction error.** Mean squared error on game outcome predictions for neural networks trained by reinforcement learning (i.e. *AlphaGo Zero*) or supervised learning respectively. For supervised learning, the network was trained for 3 days on KGS data (amateur games); comparative results are also shown from [108]. For reinforcement learning, the 20 block network was trained for 3 days and the 40 block network was trained for 40 days. Networks were also evaluated on a validation set based on professional games from the GoKifu data set.

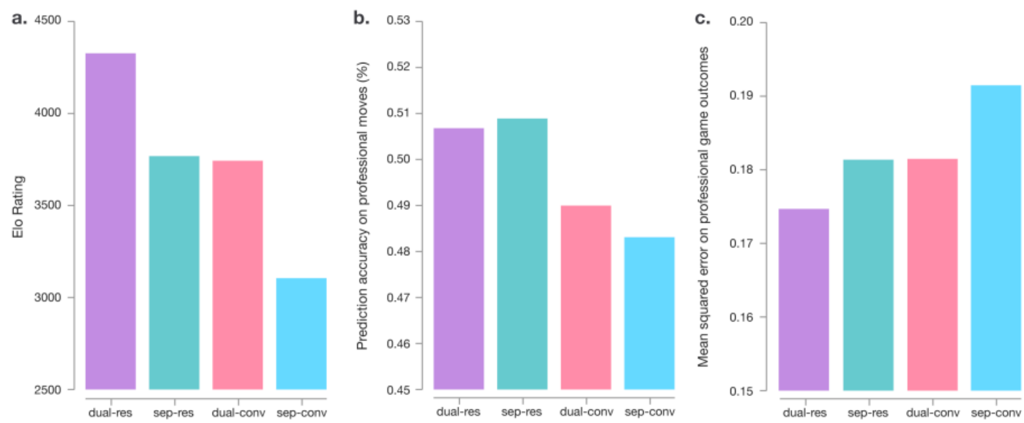


Figure 5.5: **Comparison of neural network architectures in *AlphaGo Zero* and *AlphaGo Lee*.** Comparison of neural network architectures using either separate ("sep") or combined policy and value networks ("dual"), and using either convolutional ("conv") or residual networks ("res"). The combinations "dual-res" and "sep-conv" correspond to the neural network architectures used in *AlphaGo Zero* and *AlphaGo Lee* respectively. Each network was trained on a fixed data-set generated by a previous run of *AlphaGo Zero*. **a** Each trained network was combined with *AlphaGo Zero*'s search to obtain a different player. Elo ratings were computed from evaluation games between these different players, using 5 seconds of thinking time per move. **b** Prediction accuracy on human professional moves (from the GoKifu data-set) for each network architecture. **c** Mean-squared error on human professional game outcomes (from the GoKifu data-set) for each network architecture.

accuracy compared to prior work [108, 79, 24, 128, 21] as shown in tables 5.1 and 5.2. Supervised learning achieved better initial performance, and was better at predicting the outcome of human professional games (see figure 5.3). Notably, although supervised learning achieved higher move prediction accuracy, the self-learned player performed much better overall, defeating the human-trained player within the first 24 hours of training. This suggests that *AlphaGo Zero* may be learning a strategy that is qualitatively different to human play.

To separate the contributions of architecture and algorithm, we compared the

performance of the neural network architecture in *AlphaGo Zero* with the previous neural network architecture used in *AlphaGo Lee* (see figure 5.5). Four neural networks were created, using either separate policy and value networks, as in *AlphaGo Lee*, or combined policy and value networks, as in *AlphaGo Zero*; and using either the convolutional network architecture from *AlphaGo Lee* or the residual network architecture from *AlphaGo Zero*. Each network was trained to minimise the same loss function (see equation 5.1) using a fixed data-set of self-play games generated by *AlphaGo Zero* after 72 hours of self-play training. Using a residual network was more accurate, achieved lower error, and improved performance in *AlphaGo* by over 600 Elo. Combining policy and value together into a single network slightly reduced the move prediction accuracy, but reduced the value error and boosted playing performance in *AlphaGo* by around another 600 Elo. This is partly due to improved computational efficiency, but more importantly the dual objective regularises the network to a common representation that supports multiple use cases.

Knowledge Learned by *AlphaGo Zero*

AlphaGo Zero discovered a remarkable level of Go knowledge during its self-play training process. This included fundamental elements of human Go knowledge, and also non-standard strategies beyond the scope of traditional Go knowledge. Figure 5.6 **a** shows a timeline indicating when professional joseki (corner sequences) were discovered; ultimately *AlphaGo Zero* preferred new joseki variants that were previously unknown (see figure 5.6 **b**. Figure 5.6 **c** shows fast self-play games played at different stages of training. Tournament length games played at regular intervals throughout training are shown in the appendix A. *AlphaGo Zero* rapidly progressed from entirely random moves towards a sophisticated understanding of Go concepts including fuseki (opening), tesuji (tactics), life-and-death, ko (repeated board situations), yose (endgame), capturing races, sente (initiative), shape, influence and territory, all discovered from first principles. Surprisingly, shicho (ladder capture sequences that may span the whole board) – one of the first elements of Go knowledge learned by humans – were only understood by *AlphaGo Zero* much later in training.

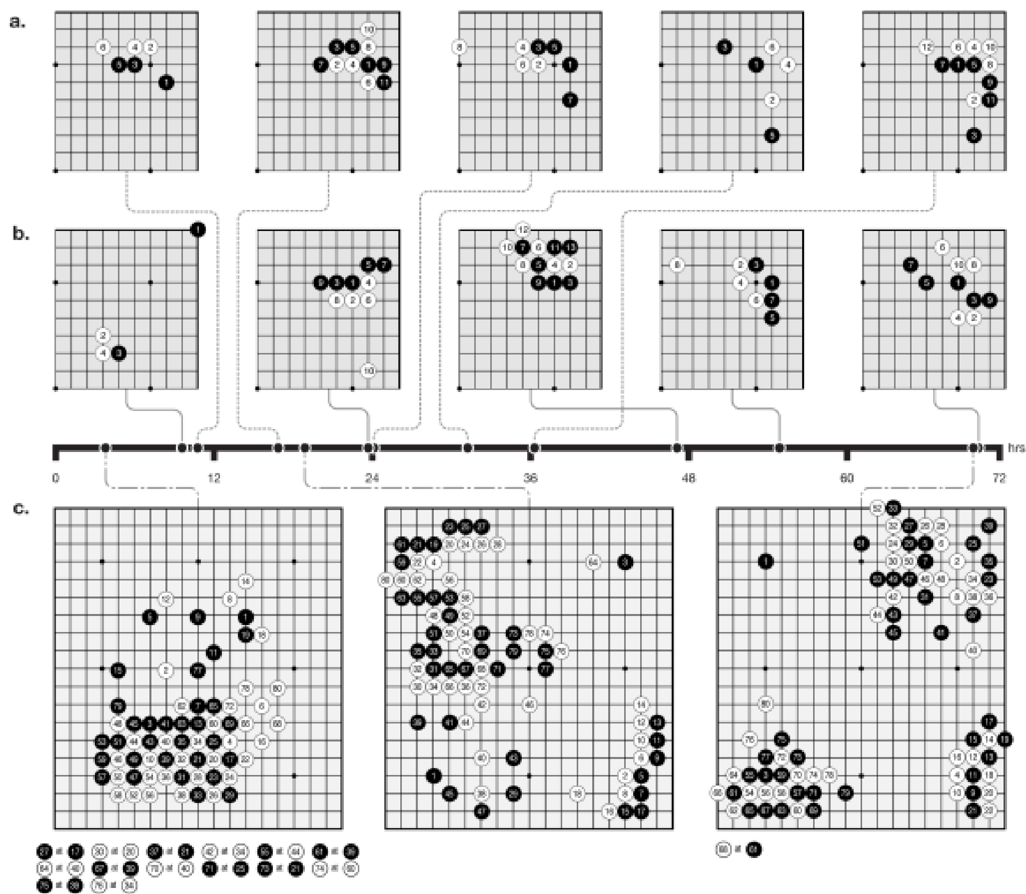


Figure 5.6: **Go knowledge learned by *AlphaGo Zero*.** **a** Five human joseki (common corner sequences) discovered during *AlphaGo Zero* training. The associated timestamps indicate the first time each sequence occured (taking account of rotation and reflection) during self-play training. **b** Five joseki favoured at different stages of self-play training. Each displayed corner sequence was played with the greatest frequency, among all corner sequences, during an iteration of self-play training. The timestamp of that iteration is indicated on the timeline. At 10 hours a weak corner move was preferred. At 47 hours the 3-3 invasion was most frequently played. This joseki is also common in human professional play; however *AlphaGo Zero* later discovered and preferred a new variation. **c** The first 80 moves of three self-play games that were played at different stages of training, using 1600 simulations (around 0.4s) per search. At 3 hours, the game focuses greedily on capturing stones, much like a human beginner. At 19 hours, the game exhibits the fundamentals of life-and-death, influence and territory. At 70 hours, the game is beautifully balanced, involving multiple battles and a complicated ko fight, eventually resolving into a half-point win for white.

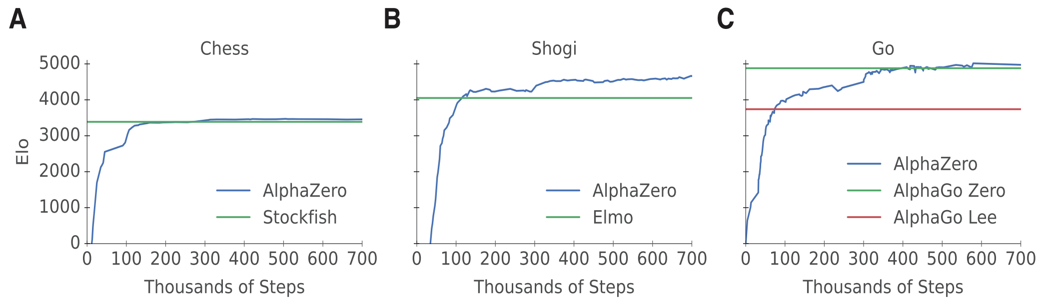


Figure 5.7: **Training *AlphaZero* for 700,000 steps.** Elo ratings were computed from games between different players where each player was given one second per move. **(A)** Performance of *AlphaZero* in chess, compared with the 2016 TCEC world-champion program *Stockfish*. **(B)** Performance of *AlphaZero* in shogi, compared with the 2017 CSA world-champion program *Elmo*. **(C)** Performance of *AlphaZero* in Go, compared with *AlphaGo Lee* and *AlphaGo Zero* (20 blocks over 3 days).

5.3.2 Results

Given the remarkable performance of our *AlphaGo Zero* agent, we focused on extending it to other domains. We removed the domain specific assumptions that *AlphaGo Zero* makes regarding the symmetries in the game and we applied our method to the games of chess, shogi and Go. Our algorithm, *AlphaZero*, achieved state-of-the-art performance across all games.

We trained separate instances of *AlphaZero* for chess, shogi and Go. Figure 5.7 shows the performance of *AlphaZero* during self-play reinforcement learning, as a function of training steps, on an Elo [26] scale [2]. Training proceeded for 700,000 steps (in mini-batches of 4,096 training positions) starting from randomly initialized parameters. During training only, 5,000 first-generation tensor processing units (TPUs) [65] were used to generate self-play games, and 16 second-generation TPUs were used to train the neural networks. Training lasted for approximately 9 hours in chess, 12 hours in shogi and 13 days in Go (see table 5.3).

In chess, *AlphaZero* first outperformed *Stockfish* after just 4 hours (300,000

[2]The prevalence of draws in high-level chess tends to compress the Elo scale, compared to shogi or Go.

	Chess	Shogi	Go
Mini-batches	700k	700k	700k
Training Time	9h	12h	13d
Training Games	44 million	24 million	140 million
Thinking Time	800 sims	800 sims	800 sims
	$\sim$ 40 ms	$\sim$ 80 ms	$\sim$ 200 ms

Table 5.3: Selected statistics of *AlphaZero* training in chess, shogi and Go.

steps); in shogi, *AlphaZero* first outperformed *Elmo* after 2 hours (110,000 steps); and in Go, *AlphaZero* first outperformed *AlphaGo Lee* after 30 hours (74,000 steps).

We evaluated the fully trained instances of *AlphaZero* against *Stockfish*, *Elmo* and the previous version of *AlphaGo Zero* in chess, shogi and Go respectively (see figure 5.8). Each program was run on the hardware for which it was designed[3]: *Stockfish* and *Elmo* used 44 central processing unit (CPU) cores (as in the TCEC world championship), whereas *AlphaZero* and *AlphaGo Zero* used a single machine with four first-generation TPUs and 44 CPU cores[4]. The chess match was played against the 2016 TCEC (season 9) world champion *Stockfish*. The shogi match was played against the 2017 CSA world champion version of *Elmo*. The Go match was played against the previous version of *AlphaGo Zero*. All matches were played using time controls of 3 hours per game, plus an additional 15 seconds for each move.

In Go, *AlphaZero* defeated *AlphaGo Zero*, winning 61% of games. This demonstrates that a general approach can recover the performance of an algorithm that exploited board symmetries to generate eight times as much data.

In chess, *AlphaZero* defeated *Stockfish*, winning 155 games and losing 6 games out of 1,000.

[3] *Stockfish* is designed to exploit CPU hardware and cannot make use of GPU/TPU, whereas *AlphaZero* is designed to exploit GPU/TPU hardware rather than CPU.

[4] A first generation TPU is roughly similar in inference speed to a Titan V GPU, although the architectures are not directly comparable.

Program	Chess	Shogi	Go
AlphaZero	63k (13k)	58k (12k)	16k (0.6k)
Stockfish	58,100k (24,000k)		
Elmo		25,100k (4,600k)	
AlphaZero	1.5 GFlop	1.9 GFlop	8.5 GFlop

Table 5.4: Evaluation speed (positions/second) of *AlphaZero*, *Stockfish*, and *Elmo* in chess, shogi and Go. Evaluation speed is the average over entire games at full time controls from the initial board position (the main evaluation in Figure 5.8), standard deviations are shown in parentheses. Bottom row: Number of operations used by *AlphaZero* for one evaluation.

In shogi, *AlphaZero* defeated *Elmo*, winning 98.2% of games when playing black, and 91.2% overall.

AlphaZero searches just 60,000 positions per second in chess and shogi, compared with 60 million for *Stockfish* and 25 million for *Elmo* (table 5.4). *AlphaZero* may compensate for the lower number of evaluations by using its deep neural network to focus much more selectively on the most promising variations. *AlphaZero* also defeated *Stockfish* when given $1/10$ as much thinking time as its opponent (i.e. searching ~ $1/10,000$ as many positions), and won 46% of games against *Elmo* when given $1/100$ as much time (i.e. searching ~ $1/40,000$ as many positions). The high performance of *AlphaZero*, using MCTS, calls into question the widely held belief [4, 32] that alpha-beta search is inherently superior in these domains.

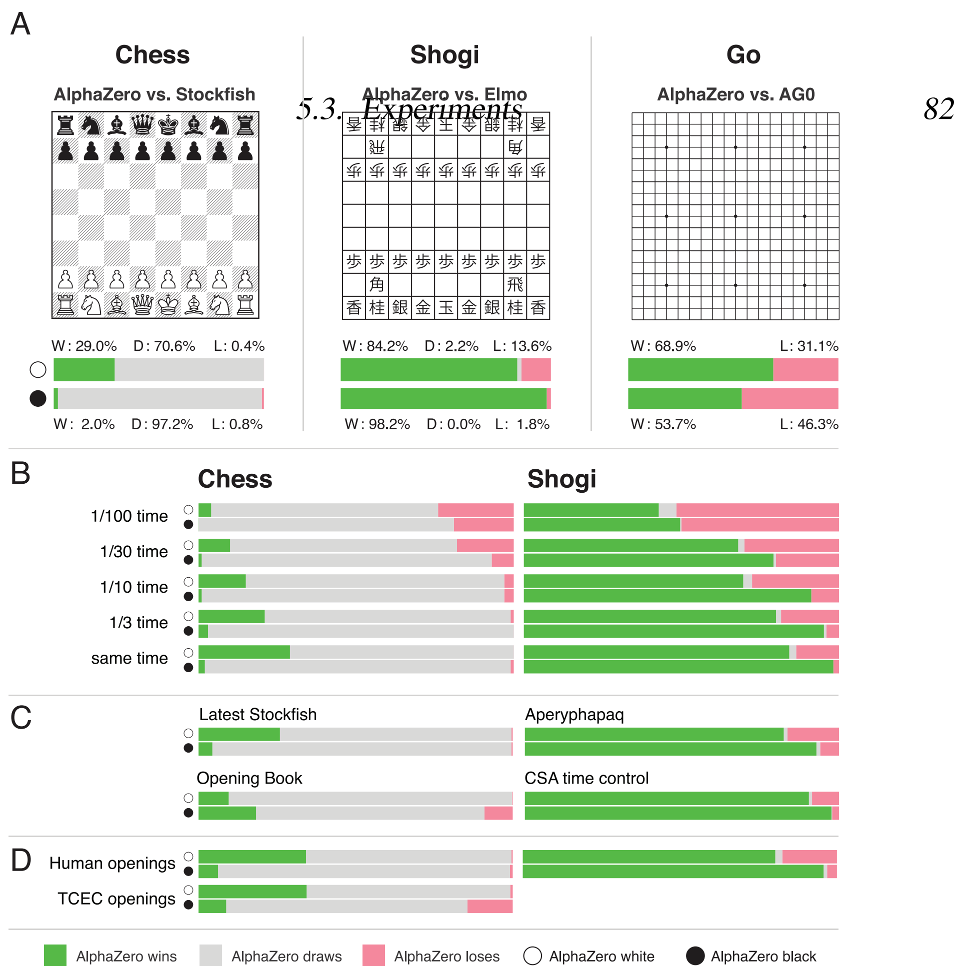


Figure 5.8: **Comparison with specialized programs.** **(A)** Tournament evaluation of *AlphaZero* in chess, shogi, and Go in matches against respectively *Stockfish*, *Elmo* and of *AlphaGo Zero* that was trained for 3 days. In the top bar, *AlphaZero* plays white; in the bottom bar *AlphaZero* plays black. Each bar shows the results from *AlphaZero*'s perspective: win ('W', green), draw ('D', grey), loss ('L', red). **(B)** Scalability of *AlphaZero* with thinking time, compared to *Stockfish* and *Elmo*. *Stockfish* and *Elmo* always receive full time (3 hours per game plus 15 seconds per move), time for *AlphaZero* is scaled down as indicated. **(C)** Extra evaluations of *AlphaZero* in chess against the most recent version of *Stockfish* at the time of writing, and against *Stockfish* with a strong opening book. Extra evaluations of *AlphaZero* in shogi were carried out against another strong shogi program *Aperyqhapaq* at full time controls and against *Elmo* under 2017 CSA world championship time controls (10 minutes per game plus 10 seconds per move). **(D)** Average result of chess matches starting from different opening positions: either common human positions, or the 2016 TCEC world championship opening positions. Average result of shogi matches starting from common human positions. CSA world championship games start from the initial board position. Match conditions are provided in appendix A.

5.3.3 Ablations

Repeatability

To measure the robustness of our method we repeated our chess experiments multiple times. Figure 5.9 shows that our training algorithm achieved similar performance in all independent runs, suggesting that the high performance of *AlphaZero*'s training algorithm is repeatable.

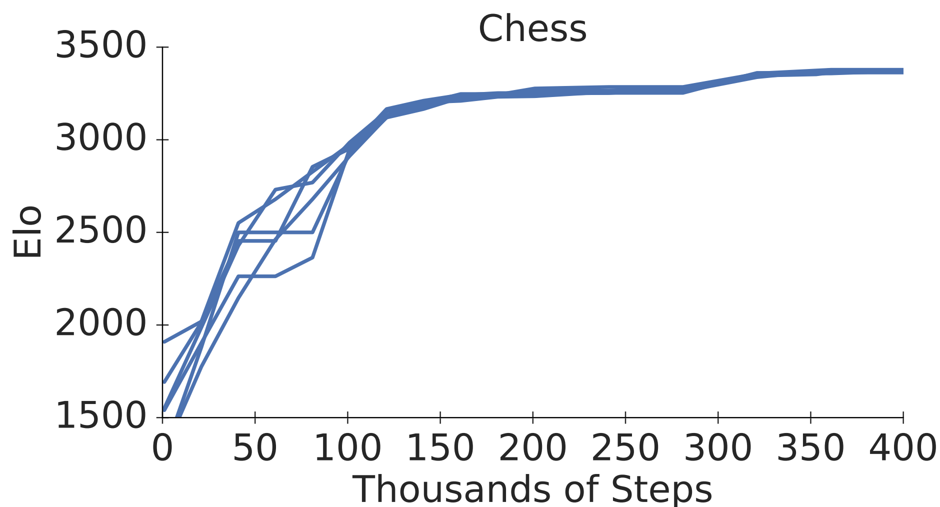


Figure 5.9: **Repeatability of *AlphaZero* training on the game of chess.** The figure shows 6 separate training runs of 400,000 steps (approximately 4 hours each). Elo ratings were computed from a tournament between baseline players and *AlphaZero* players at different stages of training. *AlphaZero* players were given 800 simulations per move. Similar repeatability was observed in shogi and Go.

Evaluations

We compared the performance of the *AlphaZero* and *AlphaGo Zero* algorithms during training. Figure 5.10 shows the Elo achieved by the two agents as a function of training steps and wall time. Since *AlphaZero* does not exploit the symmetries of the Go board during training it requires 8 times more data to achieve the same performance. We also evaluated a version of *AlphaZero* that uses the same domain knowledge as *AlphaGo Zero*. This agent outperformed *AlphaGo Zero* both in terms of training steps and wall-time, we attribute this to the superior network training

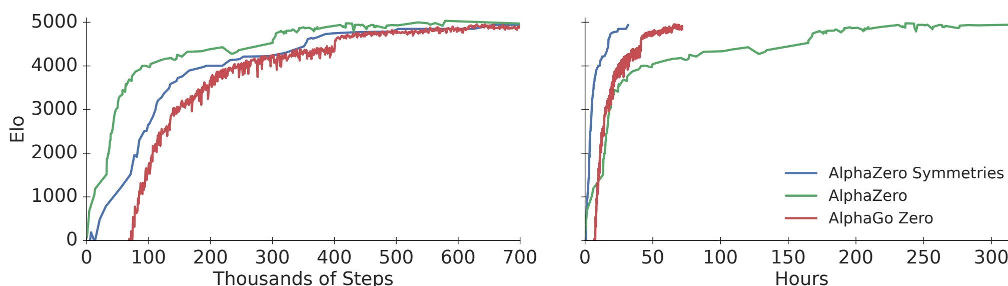


Figure 5.10: **Learning curves showing the Elo performance during training in Go.** Comparison between *AlphaZero*, a version of *AlphaZero* that exploits knowledge of symmetries in a similar manner to *AlphaGo Zero*, and the previously published *AlphaGo Zero*. *AlphaZero* generates approximately $1/8$ as many positions per training step, and therefore uses eight times more wall clock time, than the symmetry-augmented algorithms.

setup of the *AlphaZero* algorithm (see A for details).

To verify the robustness of *AlphaZero* in chess, we played additional matches that started from common human openings (see figure A). *AlphaZero* defeated *Stockfish* in each opening, suggesting that *AlphaZero* has mastered a wide spectrum of chess play. *AlphaZero* discovered and played frequently common human openings during self-play training. We also played a match that started from the set of opening positions used in the 2016 TCEC world championship; *AlphaZero* won convincingly in this match too[5] (see appendix A). We played additional matches against the most recent development version of *Stockfish* [6], and a variant of *Stockfish* that uses a strong opening book [7]. *AlphaZero* won all matches by a large margin.

[5]Many TCEC opening positions are unbalanced according to both *AlphaZero* and *Stockfish*, resulting in more losses for both players.

[6]Newest available version of *Stockfish* as of 13th of January 2018, from `https://github.com/official-stockfish/Stockfish/commit/b508f9561cc2302c129efe8d60f201ff03ee72c8`

[7]Cerebellum opening book from `https://zipproth.de/Brainfish_download`. *AlphaZero* did not use an opening book. To ensure diversity against a deterministic opening book, *AlphaZero* used a small amount of randomization in its opening moves; this avoided duplicate games but also resulted in more losses.

5.4 Conclusions

Board games have been heavily studied in the field of artificial intelligence. Games like chess and subsequently Go have acted as standard testbeds for the development of new methods. Previous state-of-the-art approaches were based on the use of expert human data and powerful engines which search many millions of positions, leveraging handcrafted domain expertise and sophisticated domain adaptations. On the other hand, *AlphaZero* has demonstrated that a pure reinforcement learning approach is fully feasible, even in the most challenging of domains: it is possible to train to superhuman level, without human examples or guidance, given no knowledge of the domain beyond basic rules. *AlphaZero*, as our Go experiments have demonstrated, required just a few more hours to train, and achieved much better asymptotic performance, compared to training on human expert data. At the same time it required a search budget of $1/1,000$ as many positions as previous programs which have dominated the research fields of computer chess and shogi. These results bring us a step closer to fulfilling a longstanding ambition of artificial intelligence [93]: a general games playing system that can learn to master any game.

Chapter 6

MuZero

As we saw previously, *AlphaZero* manages to achieve superhuman performance in challenging domains, such as chess and Go, completely from scratch by combining an MCTS search with deep neural networks. However, *AlphaZero* assumes access to a perfect simulator of the environment inside its search. This limits its applicability to domains where such a simulator is available or can be easily constructed. In many real-world problems the dynamics governing the environment are often complex and unknown. In this chapter, we present a new method, called *MuZero*[1], which, by combining a learned model with an MCTS search, achieves superhuman performance in a range of complex environments without assuming any knowledge of the underlying dynamics. When evaluated on 57 different Atari games - the canonical video game environment for testing AI techniques, in which model-based planning approaches have historically struggled - our new algorithm achieved a new state of the art. When evaluated on Go, chess and shogi, without any knowledge of the game rules, *MuZero* matched the superhuman performance of the *AlphaZero* algorithm that was supplied with the game rules.

6.1 Introduction

MuZero builds upon *AlphaZero*'s powerful search and search-based policy iteration algorithm and it extends it in two important ways: it removes the need for an explicit simulator by introducing a learned dynamics model and it modifies

[1]The *MuZero* algorithm was previously published in [103]

AlphaZero's planning and learning rules to support a broader set of environments including single agent domains and non-zero rewards at intermediate time-steps.

The main idea of the algorithm is to predict those aspects of the future that are directly relevant for planning. The model receives the observation (e.g. a Go board position or an Atari screen) as an input and transforms it into a hidden state. The hidden state is then updated iteratively by a recurrent process that receives the previous hidden state and a hypothetical next action. At every one of these steps the model produces a policy (predicting the move to play), value function (predicting the cumulative reward, for example the final game outcome), and immediate reward prediction (for example the points scored by playing a move). The model is trained end-to-end, with the sole objective of accurately estimating these three important quantities, so as to match the improved policy and value function generated by search, as well as the observed reward. There is no direct requirement or constraint on the hidden state to capture all information necessary to reconstruct the original observation, drastically reducing the amount of information the model has to maintain and predict; nor is there any requirement for the hidden state to match the unknown, true state of the environment; nor any other constraints on the semantics of state. Instead, the hidden states are free to represent any state that correctly estimates the policy, value function and reward. Intuitively, the agent can invent, internally, any dynamics that lead to accurate planning.

MuZero is designed for a more general setting than *AlphaZero*. In *AlphaZero* the planning process makes use of a *simulator* that samples the next state and reward (e.g. according to the environment's dynamics, or the rules of the game). The simulator updates the state of the game while traversing the search tree (see Figure 1 A). The simulator is used to provide three important pieces of knowledge: (1) state transitions in the search tree, (2) actions available at each node of the search tree, (3) episode termination within the search tree. In *MuZero*, all of these have been replaced with the use of a single implicit model learned by a neural network (see Figure 1 B):

- **State transitions**. *AlphaZero* had access to a perfect simulator of the environment's dynamics. In contrast, *MuZero* employs a learned dynamics model within its search. Under this model, each node in the tree is represented by a corresponding hidden state; by providing a hidden state s_{k-1} and an action a_k to the model the search algorithm can transition to a new node $s_k = g(s_{k-1}, a_k)$.

- **Legal actions**. We consider a standard problem formulation where the set of legal actions is provided at each time-step alongside the observation. During search, however, it could be helpful to specify the legal actions at each interior node - which would require knowledge of how the legal actions change over time. *AlphaZero* used the set of legal actions obtained from the simulator to mask the policy network at interior nodes. *MuZero* does not perform any masking within the search tree, but only masks legal actions at the root of the search tree where the set of legal actions is directly observed. The policy network rapidly learns to exclude actions that are unavailable, simply because they are never selected.

- **Terminal states**. *AlphaZero* stopped the search at tree nodes representing terminal states and used the terminal value provided by the simulator instead of the value produced by the network. *MuZero* does not give special treatment to terminal states and always uses the value predicted by the network. Inside the tree, the search can proceed past a state that would terminate the simulator. In this case the network is expected to always predict the same value, which may be achieved by modelling terminal states as absorbing states during training.

In addition, *MuZero* is designed to operate in the general reinforcement learning setting: single-agent domains with discounted intermediate rewards of arbitrary magnitude. In contrast, *AlphaZero* was designed to operate in two-player games with undiscounted terminal rewards of ± 1.

6.2 Algorithm

In this section we describe the *MuZero* algorithm in detail. A schematic illustration of our method is shown in Figure 1.

6.2.1 Model

The *MuZero* model makes predictions at each time-step t, for each of $k = 0...K$ steps, by a model μ_θ, with parameters θ, conditioned on past observations $o_1, ..., o_t$ and for $k > 0$ on future actions $a_{t+1}, ..., a_{t+k}$. The model predicts three future quantities: the policy $p_t^k \approx \pi(a_{t+k+1}|o_1, ..., o_t, a_{t+1}, ..., a_{t+k})$, the value function $v_t^k \approx \mathbb{E}[u_{t+k+1} + \gamma u_{t+k+2} + ...|o_1, ..., o_t, a_{t+1}, ..., a_{t+k}]$, and for $k > 0$ also the immediate reward $r_t^k \approx u_{t+k}$, where $u_.$ is the true, observed reward, π is the policy used to select real actions, and γ is the discount function of the environment.

Internally, at each time-step t (subscripts $_t$ suppressed for simplicity), the model is represented by the combination of a *representation* function, a *dynamics* function, and a *prediction* function. The *MuZero* model parts are shown in equation 6.1.

$$\begin{aligned} \textit{Representation} \quad & s^0 = h_\theta(o_1, ..., o_t) \\ \textit{Dynamics} \quad & r^k, s^k = g_\theta(s^{k-1}, a^k) \\ \textit{Prediction} \quad & p^k, v^k = f_\theta(s^k) \end{aligned} \tag{6.1}$$

The dynamics function, $r^k, s^k = g_\theta(s^{k-1}, a^k)$, is a recurrent process that computes, at each hypothetical step k, an immediate reward r^k and an internal state s^k. It mirrors the structure of an MDP model that computes the expected reward and state transition for a given state and action [96]. However, unlike traditional approaches to model-based RL [117], this internal state s^k has no semantics of environment state attached to it – it is simply the hidden state of the overall model, and its sole purpose is to accurately predict relevant, future quantities: policies, values, and rewards. The policy and value functions are computed from the internal state s^k by the prediction function, $p^k, v^k = f_\theta(s^k)$, akin to the joint policy and value network of *AlphaZero*. The "root" state s^0 is initialized using a representation function that encodes past observations, $s^0 = h_\theta(o_1, ..., o_t)$; again this has no special semantics beyond its support for future predictions.

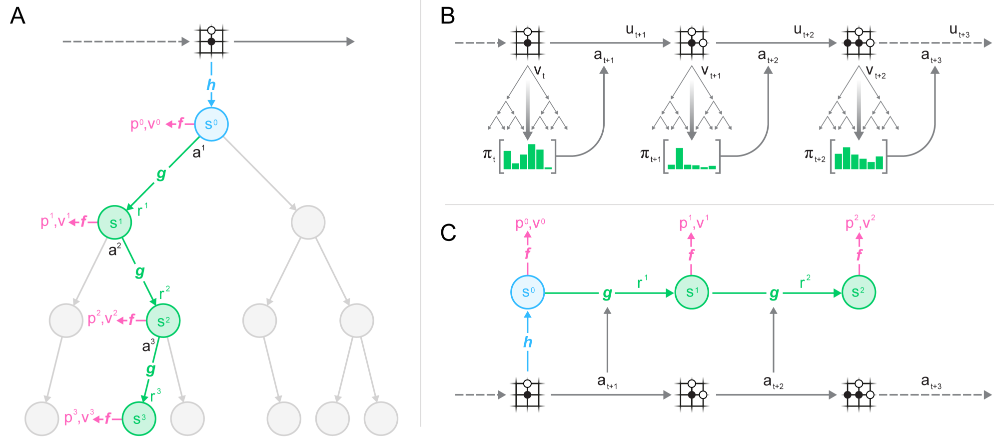


Figure 1: **Planning, acting, and training with a learned model. (A)** How *MuZero* uses its model to plan. The model consists of three connected components for representation, dynamics and prediction. Given a previous hidden state s^{k-1} and a candidate action a^k, the *dynamics* function g produces an immediate reward r^k and a new hidden state s^k. The policy p^k and value function v^k are computed from the hidden state s^k by a *prediction* function f. The initial hidden state s^0 is obtained by passing the past observations (e.g. the Go board or Atari screen) into a *representation* function h. **(B)** How *MuZero* acts in the environment. A Monte-Carlo Tree Search is performed at each timestep t, as described in A. An action a_{t+1} is sampled from the search policy π_t, which is proportional to the visit count for each action from the root node. The environment receives the action and generates a new observation o_{t+1} and reward u_{t+1}. At the end of the episode the trajectory data is stored into a replay buffer. **(C)** How *MuZero* trains its model. A trajectory is sampled from the replay buffer. For the initial step, the representation function h receives as input the past observations $o_1, ..., o_t$ from the selected trajectory. The model is subsequently unrolled recurrently for K steps. At each step k, the dynamics function g receives as input the hidden state s^{k-1} from the previous step and the real action a_{t+k}. The parameters of the representation, dynamics and prediction functions are jointly trained, end-to-end by backpropagation-through-time, to predict three quantities: the policy $p^k \approx \pi_{t+k}$, value function $v^k \approx z_{t+k}$, and reward $r^k \approx u_{t+k}$, where z_{t+k} is a sample return: either the final reward (board games) or n-step return (Atari).

6.2.2 Search

Given such a model, it is possible to search over hypothetical future trajectories $a^1, ..., a^k$ given past observations $o_1, ..., o_t$. For example, a naive search could simply select the k step action sequence that maximizes the value function. More generally, we may apply any MDP planning algorithm to the internal rewards and state space induced by the dynamics function. Specifically, we use an MCTS algorithm similar to *AlphaZero*'s search, generalized to allow for single agent domains and intermediate rewards. The MCTS algorithm may be viewed as a *search policy* $\pi_t = \mathbb{P}[a_{t+1}|o_1, ..., o_t]$ and *search value function* $v_t \approx \mathbb{E}[u_{t+1} + \gamma u_{t+2} + ...|o_1, ..., o_t]$ that both selects an action and predicts cumulative reward given past observations $o_1, ..., o_t$. At each internal node, it makes use of the policy, value function and reward estimate produced by the current model parameters θ, and combines these values together using lookahead search to produce an improved policy π_t and improved value function v_t at the root of the search tree. The next action $a_{t+1} \sim \pi_t$ is then chosen by the search policy.

MuZero modifies *AlphaZero*'s MCTS search to support planning with its learned model and in domains other than two player board games, in the following ways:

- The statistics stored at each edge of the tree are augmented to include the reward r^k and state s^k which were generated by the model when that edge was first expanded.

- The backup is generalized to the case where the environment can emit intermediate rewards, have a discount γ different from 1, and the value estimates are unbounded. For $k = l...0$, we form an $l-k$-step estimate of the cumulative discounted reward, bootstrapping from the value function v^l,

$$G^k = \sum_{\tau=0}^{l-1-k} \gamma^\tau r_{k+1+\tau} + \gamma^{l-k} v^l \tag{6.2}$$

- In environments where the value is unbounded the pUCT rule is adjusted. The Q value estimates used in the pUCT formula, are normalized using the minimum-maximum values observed in the search tree up to that point.

Those changes allow the application of the *MuZero* search in any MDP environment without requiring any domain specific knowledge about the transition dynamics or the reward function.

6.2.3 Self-play

Similarly to *AlphaZero*, *MuZero* generates experience by selecting actions based on a policy produced by an MCTS search at each time-step t. The search is guided by the current version of *MuZero*'s learned model. Subsequently, the observed transitions are stored in a common replay buffer.

6.2.4 Training

During training, all parameters of the model are trained jointly to accurately match the policy, value function and reward prediction, for every hypothetical step k, to three corresponding targets observed after k actual time-steps have elapsed. Similarly to *AlphaZero*, the first objective is to minimize the error between the actions predicted by the policy p_t^k and by the search policy π_{t+k}. Also like *AlphaZero*, value targets are generated by playing out the game or MDP using the search policy. However, unlike *AlphaZero*, we allow for long episodes with discounting and intermediate rewards by *bootstrapping n* steps into the future from the search value, $z_t = u_{t+1} + \gamma u_{t+2} + ... + \gamma^{n-1} u_{t+n} + \gamma^n \nu_{t+n}$. Final outcomes $\{lose, draw, win\}$ in board games are treated as rewards $u_t \in \{-1, 0, +1\}$ occurring at the final step of the episode. Specifically, the second objective is to minimize the error between the value function v_t^k and the value target, z_{t+k}.[2] The third objective is to minimize the error between the predicted immediate reward r_t^k and the observed immediate reward u_{t+k}. Finally, an L2 regularization term is also added, leading to the overall loss:

[2]For chess, Go and shogi, the same squared error loss as *AlphaZero* is used for rewards and values. A cross-entropy loss was found to be more stable than a squared error when encountering rewards and values of variable scale in Atari. Cross-entropy was used for the policy loss in both cases.

$$l_t(\theta) = \sum_{k=0}^{K} l^p(\pi_{t+k}, p_t^k) + \sum_{k=0}^{K} l^v(z_{t+k}, v_t^k) + \sum_{k=1}^{K} l^r(u_{t+k}, r_t^k) + c||\theta||^2 \qquad (6.3)$$

where l^p, l^v, and l^r are loss functions for policy, value and reward respectively. Those losses are instantiated differently depending on the environment (see B.6).

6.2.5 Reanalyze

One of the advantages of model-based reinforcement learning is that it can leverage its model to improve the data efficiency of its learning rule. The model can generate synthetic experience which can be used to obtain better value estimates and policies. In order to reap those benefits, we developed a second variant of the *MuZero* algorithm, called *MuZero Reanalyze*. *MuZero Reanalyze* revisits its past trajectories and re-executes its search using the latest model parameters, thus producing a better quality policy than the original search. This fresh policy can then be used as a new policy target when updating the *MuZero* model. The search also provides new better value estimates which can be utilized during training. However, in our implementation, we empirically found that it is preferably to use bootstrap value estimates computed directly by the *representation* function of the model using a target network mechanism[84]. The target network is a copy of the *representation* network where its parameters are updated every N steps to match those of the online network. By keeping the parameters constant for a number of steps during training we improve learning by reducing the non-stationarity of the value targets.

6.3 Experiments

6.3.1 Results

We applied the *MuZero* algorithm to the classic board games of Go, chess and shogi, as benchmarks for challenging planning problems, and to all 57 games in the Atari Learning Environment [15], as benchmarks for visually complex RL domains. In each case we trained *MuZero* for $K = 5$ hypothetical steps. Training proceeded for 1 million mini-batches of size 2048 in board games and of size 1024 in Atari. During both training and evaluation, *MuZero* used 800 simulations for each search in board

games, and 50 simulations for each search in Atari. The representation function uses the same convolutional [75] and residual [50] architecture as *AlphaZero*, but with 16 residual blocks instead of 20. The dynamics function uses the same architecture as the representation function and the prediction function uses the same architecture as *AlphaZero*. All networks use 256 hidden planes.

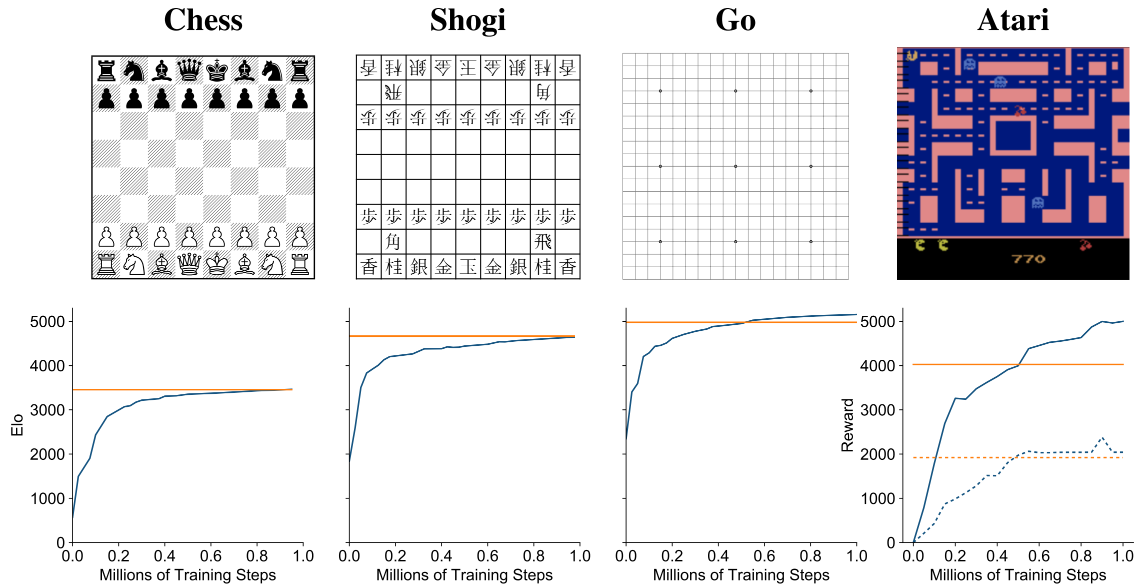


Figure 2: **Evaluation of *MuZero* throughout training in chess, shogi, Go and Atari.** The x-axis shows millions of training steps. For chess, shogi and Go, the y-axis shows Elo rating, established by playing games against *AlphaZero* using 800 simulations per move for both players. *MuZero*'s Elo is indicated by the blue line, *AlphaZero*'s Elo by the horizontal orange line. For Atari, mean (full line) and median (dashed line) human normalized scores across all 57 games are shown on the y-axis. The scores for R2D2 [68], (the previous state of the art in this domain, based on model-free RL) are indicated by the horizontal orange lines. Performance in Atari was evaluated using 50 simulations every fourth time-step, and then repeating the chosen action four times, as in prior work [84].

Figure 2 shows the performance throughout training in each game. In Go, *MuZero* slightly exceeded the performance of *AlphaZero*, despite using less computation per node in the search tree (16 residual blocks per evaluation in *MuZero*

Agent	Median	Mean	Env. Frames	Training Time	Training Steps
Ape-X [55]	434.1%	1695.6%	22.8B	5 days	8.64M
R2D2 [68]	1920.6%	4024.9%	37.5B	5 days	2.16M
MuZero	**2041.1%**	**4999.2%**	20.0B	12 hours	1M
IMPALA [36]	191.8%	957.6%	200M	–	–
Rainbow [53]	231.1%	–	200M	10 days	–
UNREAL[a] [61]	250%[a]	880%[a]	250M	–	–
LASER [102]	431%	–	200M	–	–
MuZero Reanalyze	**741.7%**	**2183.6%**	200M	12 hours	1M

Table 1: **Comparison of *MuZero* against previous agents in Atari**. We compare separately against agents trained in large (top) and small (bottom) data settings; all agents other than *MuZero* used model-free RL techniques. Mean and median scores are given, compared to human testers. The best results are highlighted in **bold**. *MuZero* sets a new state of the art in both settings. [a]Hyper-parameters were tuned per game.

compared to 20 blocks in *AlphaZero*). This suggests that *MuZero* may be caching its computation in the search tree and using each additional application of the dynamics model to gain a deeper understanding of the position.

In Atari, *MuZero* achieved a new state of the art for both mean and median normalized score across the 57 games of the Arcade Learning Environment, outperforming the previous state-of-the-art method R2D2 [68] (a model-free approach) in 42 out of 57 games, and outperforming the previous best model-based approach SimPLe [66] in all games. *MuZero Reanalyze* was also evaluated on the Atari 57 suite using 200 million frames of experience per game, achieving a median normalized score of 731% and outperforming previous state-of-the-art model-free approaches. Table 1 summarizes the results of our Atari experiments.

Details regarding the implementation of our algorithm and the hyperparameters used can be found the in the appendix B.

6.3.2 Ablations

To understand the role of the model in *MuZero* we ran several ablation experiments, focusing on the board game of Go and the Atari game of Ms. Pacman.

First, we tested the scalability of planning (Figure 3A), in the canonical planning problem of Go. We compared the performance of search in *AlphaZero*, using a perfect model, to the performance of search in *MuZero*, using a learned model. Specifically, the fully trained *AlphaZero* or *MuZero* was evaluated by comparing MCTS with different thinking times. *MuZero* matched the performance of a perfect model, even when doing much larger searches (up to 10s thinking time) than those from which the model was trained (around 0.1s thinking time, see also Figure 4A).

We also investigated the scalability of planning across all Atari games (see Figure 3B). We compared MCTS with different numbers of simulations, using the fully trained *MuZero*. The improvements due to planning are much less marked than in Go, perhaps because of greater model inaccuracy; performance improved slightly with search time, but plateaued at around 100 simulations. Even with a single simulation – i.e. when selecting moves solely according to the policy network – *MuZero* performed well, suggesting that, by the end of training, the raw policy has learned to internalise the benefits of search (see also Figure 4B).

Next, we tested our model-based learning algorithm against a comparable model-free learning algorithm (see Figure 3C). We replaced the training objective of *MuZero* (Equation 1) with a model-free Q-learning objective (as used by R2D2), and the dual policy and value heads with a single head representing the Q-function $Q(\cdot|s_t)$. Subsequently, we trained and evaluated the new model without using any search. When evaluated on Ms. Pacman, our model-free algorithm achieved identical results to R2D2, but learned significantly slower than *MuZero* and converged to a much lower final score. We conjecture that the search-based policy improvement step of *MuZero* provides a stronger learning signal than the high bias, high variance targets used by Q-learning.

To better understand the nature of *MuZero*'s learning algorithm, we measured how *MuZero*'s training scales with respect to the amount of search it uses *during*

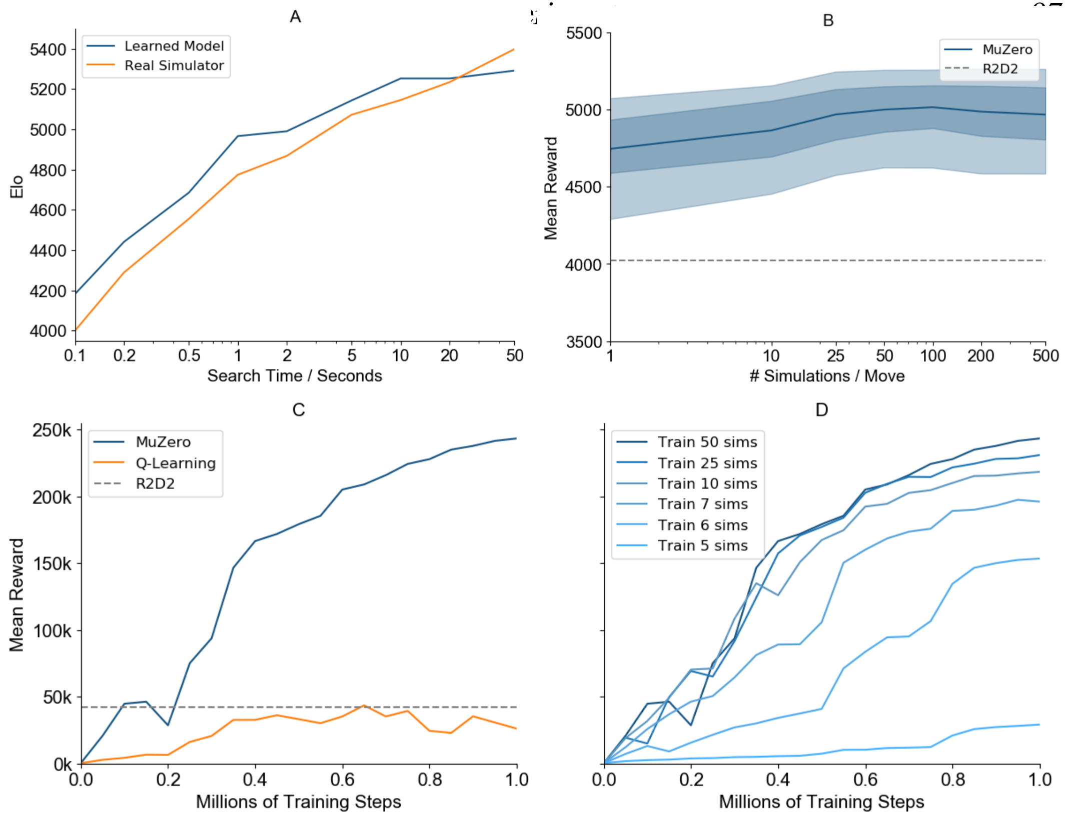


Figure 3: **Evaluations of *MuZero* on Go (A), all 57 Atari Games (B) and Ms. Pacman (C-D).** (**A**) Scaling with search time per move in Go, comparing the learned model with the ground truth simulator. Both networks were trained at 800 simulations per search, equivalent to 0.1 seconds per search. Remarkably, the learned model is able to scale well to up to two orders of magnitude longer searches than seen during training. (**B**) Scaling of final human normalized mean score in Atari with the number of simulations per search. The network was trained at 50 simulations per search. Dark line indicates mean score, shaded regions indicate 25th to 75th and 5th to 95th percentiles. The learned model's performance increases up to 100 simulations per search. Beyond, even when scaling to much longer searches than during training, the learned model's performance remains stable and only decreases slightly. This contrasts with the much better scaling in Go (A), presumably due to greater model inaccuracy in Atari than Go. (**C**) Comparison of MCTS based training with Q-learning in the *MuZero* framework on Ms. Pacman, keeping network size and amount of training constant. The state of the art Q-Learning algorithm R2D2 is shown as a baseline. Our Q-Learning implementation reaches the same final score as R2D2, but improves slower and results in much lower final performance compared to MCTS based training. (**D**) Different networks trained at different numbers of simulations per move, but all evaluated at 50 simulations per move. Networks trained with more simulations per move improve faster, consistent with ablation (B), where the policy improvement is larger when using more simulations per move. Surprisingly, *MuZero* can learn effectively even when training with less simulations per move than are enough to cover all 8 possible actions in Ms. Pacman.

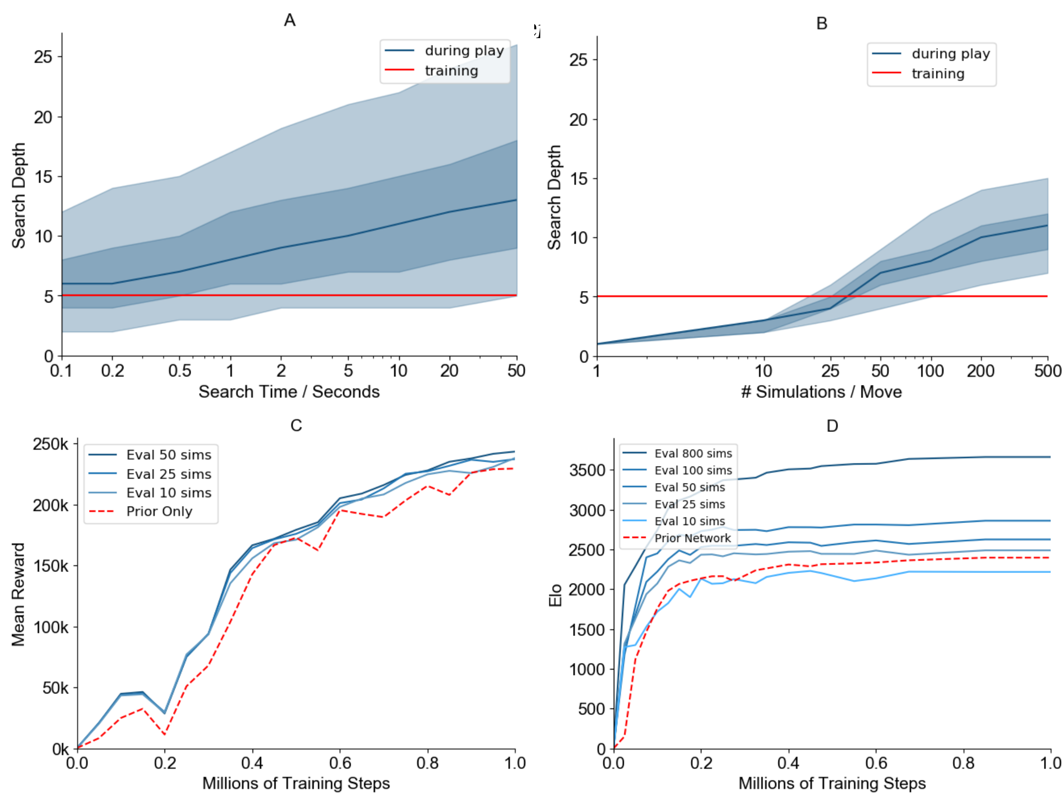


Figure 4: **Details of *MuZero* evaluations (A-B) and policy improvement ablations (C-D)**. (**A-B**) Distribution of evaluation depth in the search tree for the learned model for the evaluations in Figure 3A-B. The network was trained over 5 hypothetical steps, as indicated by the red line. Dark blue line indicates median depth from the root, dark shaded region shows 25th to 75th percentile, light shaded region shows 5th to 95th percentile. (**C**) Policy improvement in Ms. Pacman - a single network was trained at 50 simulations per search and is evaluated at different numbers of simulations per search, including playing according to the argmax of the raw policy network. The policy improvement effect of the search over the raw policy network is clearly visible throughout training. This consistent gap between the performance with and without search highlights the policy improvement that *MuZero* exploits, by continually updating towards the improved policy, to efficiently progress towards the optimal policy. (**D**) Policy improvement in Go - a single network was trained at 800 simulations per search and is evaluated at different numbers of simulations per search. In Go, the playing strength improvement from longer searches is much larger than in Ms. Pacman and persists throughout training, consistent with our previous results. This suggests, as might intuitively be expected, that the benefit of models is greatest in precision planning domains.

training. Figure 3D shows the performance in Ms. Pacman, using an MCTS of different simulation counts per move throughout training. Surprisingly, and in contrast to previous work [8], even with only 6 simulations per move – fewer than the number of actions – *MuZero* learned an effective policy and improved rapidly. With more simulations performance jumped significantly higher. For analysis of the policy improvement during each individual iteration, see also Figure 4 C and D.

6.4 Conclusions

Many of the breakthroughs in artificial intelligence have been based on either high-performance planning [20] or model-free reinforcement learning methods [84, 89, 132]. In this chapter we have described a method that combines the benefits of both approaches. Our algorithm, *MuZero*, has both matched the superhuman performance of high-performance planning algorithms in their favoured domains – logically complex board games such as chess and Go – and outperformed state-of-the-art model-free RL algorithms in their favoured domains – visually complex Atari games. Crucially, our method does not require any knowledge of the environment dynamics, potentially paving the way towards the application of powerful learning and planning methods to a host of real-world domains for which there exists no perfect simulator.

Chapter 7

Stochastic MuZero

Despite its generality and impressive performance the applicability of the *MuZero* agent is still limited by its use of deterministic models. This hinders its performance in environments that are inherently stochastic, partially observed, or so large and complex that they appear stochastic to a finite agent. In order to overcome this limitation, we developed a new algorithm, called *Stochastic MuZero*[1], which learns a stochastic model of the environment dynamics, by incorporating afterstates, and subsequently, uses this model to perform a stochastic tree search. *Stochastic MuZero* matched or exceeded the state of the art in a set of canonical single and multi-agent environments, including *2048* and backgammon, while maintaining the superhuman performance of standard *MuZero* in the game of Go. In this section we describe our new method in detail.

7.1 Introduction

Stochastic MuZero is the first empirically effective approach for handling stochasticity in value equivalent (see 2.5.1) model-learning and planning. It extends the *MuZero* model and MCTS search to account for stochasticity in the environment dynamics. Its model is factored to first transition deterministically from state to an afterstate, and then to branch stochastically from the afterstate to the next state. This factored model is trained end-to-end so as to maintain value equivalence for both state value function and action value function respectively, and is combined

[1] The *Stochastic MuZero* algorithm was previously published in [3]

with a stochastic variant of the MCTS algorithm. We implement it using a discrete generative network, and subsequently, we extend MCTS to effectively use it by introducing chance nodes in the tree.

We apply our method, *Stochastic MuZero*, to several environments in which handling stochasticity is important. First, we consider the popular stochastic puzzle game *2048*, in which the prior state of the art exploits a perfect simulator and significant handcrafted domain knowledge. In our experiments, *Stochastic MuZero* achieved better results without any domain knowledge. Secondly, we consider the classic stochastic two-player game of backgammon, in which near-optimal play has been achieved using a perfect simulator. *Stochastic MuZero* matches this performance without any prior knowledge of the game rules. Finally, we evaluated our method in the deterministic board game of Go. There our method matched the performance of *MuZero*, demonstrating that *Stochastic MuZero* extends *MuZero* without sacrificing performance.

7.2 Algorithm

In this section we present our novel algorithm *Stochastic MuZero*. Our approach combines a learned stochastic transition model of the environment dynamics with a variant of Monte Carlo tree search (MCTS). First, we describe the new model and subsequently how it is combined with MCTS for planning.

7.2.1 Model

Afterstates We consider the problem of modeling the dynamics of a stochastic environment. Similarly to *MuZero*, the model receives an initial observation $o_{\leq t}$ at time step t and a sequence of actions $a_{t:t+K}$, and needs to make predictions about the future values, policies and rewards. In contrast to *MuZero* which only considers latent states which correspond to real states of the environment, *Stochastic MuZero* makes use of the notion of afterstates [117] to capture the stochastic dynamics. An afterstate as_t is the hypothetical state of the environment after an action is applied

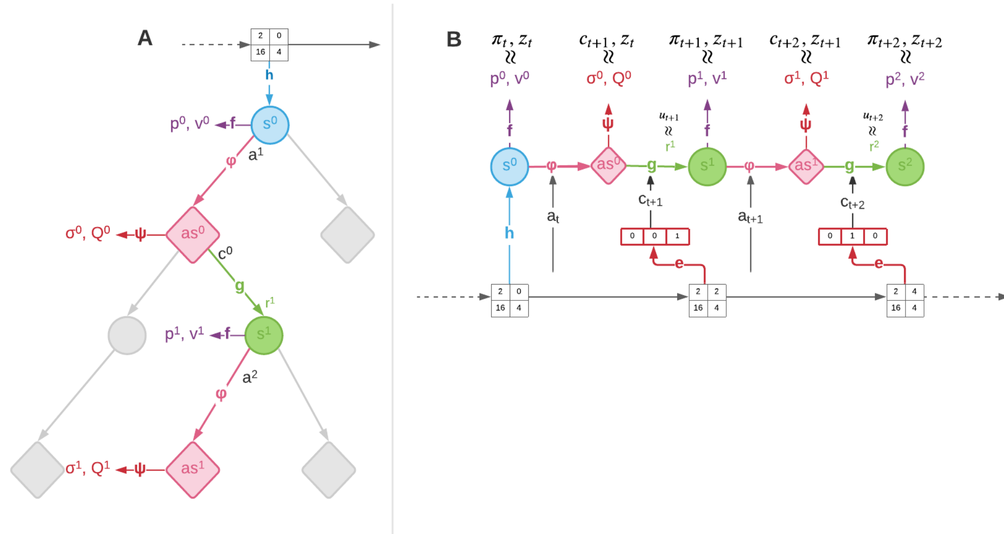
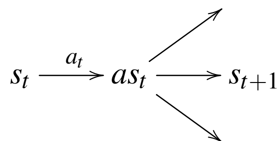


Figure 1: ***Stochastic MuZero***. **(A)** Monte Carlo Tree Search used in *Stochastic MuZero*, where diamond nodes represent *chance* nodes and circular nodes represent *decision* nodes. During the *selection* phase edges are selected by applying the pUCT formula in the case of decision nodes, and by sampling the prior σ in the case of chance nodes. **(B)** Training of stochastic model in *Stochastic MuZero*. Here for a given trajectory of length 2 with observations $o_{\leq t:t+2}$, actions $a_{t:t+2}$, value targets $z_{t:t+2}$, policy targets $\pi_{t:t+2}$ and rewards $u_{t+1:t+K}$, the model is unrolled for 2 steps. During the unroll, the encoder e receives the observation $o_{\leq t+k}$ as an input and generates a chance code c_{t+k} deterministically. The policy, value and reward outputs of the model are trained towards the targets π_{t+k}, z_{t+k} and u_{t+k} respectively. The distributions σ^k over future codes are trained to predict the code produced by the encoder.

but before the environment has transitioned to a true state:



By using afterstates, we can separate the effect of applying an action to the environment and of the chance transition given an action. For example, in backgammon, the afterstate corresponds to the board state after one player has played its action but before the other player had the chance to roll the dice. It is also possible to define the value of an afterstate as $V(as_t) = Q(s_t, a_t)$ and the transition probabilities of the environment dynamics $Pr(s_{t+1} \mid as_t) = Pr(s_{t+1} \mid s_t, a_t)$. An afterstate can lead to multiple states based on a chance event. In our work we assume that there is a finite number of possible states M that the environment can transition to, given an afterstate, and this way we can associate each transition with a chance outcome c_t^i. An example of a chance outcome could be the result of the dice in a game of backgammon. By defining afterstates as_t and chance outcomes c_t, we can model a chance transition using a deterministic model $s_{t+1}, r_{t+1} = \mathcal{M}(as_t, c_t)$ and a distribution $Pr(s_{t+1} \mid as_t) = Pr(c_t \mid as_t)$. The task of learning a stochastic model is then reduced to the problem of learning afterstates as and chance outcomes c.

Model The stochastic model of *Stochastic MuZero* consists of 5 functions: a *representation* function h which maps the current observation $o_{\leq t}$ to a latent state s_t^0, an *afterstate dynamics* function ϕ which given a state s_t^k and an action a_{t+k} produces the next latent afterstate as_t^k, a *dynamics* function g which given an afterstate as_t^k and a chance outcome c_{t+k+1} produces the next latent state s_t^{k+1} and a reward prediction r_t^{k+1}, a *prediction* function f which given a state s_t^k generates the value v_t^k and policy p_t^k predictions, and a *afterstate prediction* function ψ which given an afterstate as^k generates a value prediction Q_t^k, and a distribution $\sigma_t^k = Pr(c_{t+k+1} \mid as_t^k)$ over

possible future chance outcomes c_{t+k+1}. The model equations are shown in 7.1.

$$\begin{aligned} \textit{Representation} \quad & s_t^0 = h(o_{\leq t}) \\ \textit{Prediction} \quad & p_t^k, v_t^k = f(s_t^k) \\ \textit{Afterstate Dynamics} \quad & as_t^k = \phi(s_t^k, a_{t+k}) \\ \textit{Afterstate Prediction} \quad & \sigma_t^k, Q_t^k = \psi(as_t^k) \\ \textit{Dynamics} \quad & s_t^{k+1}, r_t^{k+1} = g(as_t^k, c_{t+k+1}) \end{aligned} \tag{7.1}$$

During inference, given an initial observation $o_{\leq t}$ and a sequence of actions $a_{t:t+K}$, we can generate trajectories from the above model by recurrently unrolling it and by sampling chance outcomes from the distributions $c_{t+k+1} \sim \sigma_t^k$.

Chance outcomes *Stochastic MuZero* models the chance outcomes by using a novel variant of the VQ-VAE method. Vector Quantised Variational AutoEncoder (VQ-VAE, [129]) is a generative modeling technique which uses four key components: an encoder neural network e, a decoder neural network d, a vector quantisation layer vq, and an autoregressive model m. Given an input x_t, the encoder produces an embedding $c_t^e = e(x_t)$. The quantisation layer comprises of a set of M codes $\{c_i\}_{i=0}^M$, called the *codebook*, and quantises the encoder's output embedding c_t^e by returning the nearest code $c_t = c_{k_t}$ along with its index $k_t = \arg\min_i \|c_i - c_t^e\|$. Additionally, in the backwards pass, this quantisation is treated as an identity function, referred to as *straight-through* gradient estimation [16]. The decoder produces a reconstruction of the input $\hat{x}_t = d(c_t)$. The autoregressive model predicts a distribution $p(k_t|c_{<t}) = m(c_{<t})$ over the code index at time t using the quantised embeddings $c_{<t}$ of the previous timesteps. The VQ-VAE equations are shown in Equations 7.2.

$$\begin{aligned} \textit{Encoder} \quad & c_t^e = e(x_t) \\ \textit{Quantisation} \quad & c_t, k_t = vq(c_t^e) \\ \textit{Decoder} \quad & \hat{x}_t = d(c_t) \\ \textit{Model} \quad & p(k_t|c_{<t}) = m(c_{<t}) \end{aligned} \tag{7.2}$$

Typically, the encoder, decoder, and codebook are trained first and then frozen to train the autoregressive model in an additional second stage. The total loss for the

VQ-VAE is

$$L_{\phi}^{vqvae} = \sum_{t=0}^{N-1} \left[\underbrace{\|\hat{x}_t - x_t\|}_{\text{reconstruction}} + \beta \underbrace{\|c_t - c_t^e\|^2}_{\text{commitment}} - \underbrace{\gamma \log p(k_t | c_{<t})}_{\text{second stage}} \right] \tag{7.3}$$

Stochastic MuZero uses a VQ-VAE with a constant codebook of size M. Each entry in the codebook is a fixed one-hot vector of size M. By using a fixed codebook of one hot vectors, we can simplify the equations of the VQ-VAE 7.2. In this case, we model the encoder embedding c_t^e as a categorical variable, and selecting the closest code c_t is equivalent to computing the expression $\text{onehot}(\arg\max_i(c_t^{e,i}))$. The resulting encoder can also be viewed as a stochastic function of the observation which makes use of the Gumbel softmax reparameterization trick [62] with zero temperature during the forward pass and a straight through estimator during the backward. There is no explicit decoder in our model, and contrary to previous work [90] we do not make use of a reconstruction loss. Instead the network is trained end-to-end in a fashion similar to *MuZero*. In section 7.2.3 we explain the training procedure in more detail.

7.2.2 Search

Stochastic MuZero extends the MCTS algorithm used in *MuZero* by introducing chance nodes and chance values to the search. In the stochastic instantiation of MCTS, there are two types of nodes: *decision* and *chance* [25]. The *chance* and *decision* nodes are interleaved along the depth of the tree, so that the parent of each *decision* node is a *chance* node. The root node of the tree is always a *decision* node. In our approach, each *chance* node corresponds to a latent *afterstate* (7.2.1) and it is expanded by querying the stochastic model, where the parent state and an action are provided as an input and the model returns a value for the node and a prior distribution over future codes $Pr(c \mid as)$. After a *chance* node is expanded, its value is backpropagated up the tree. Finally, when the node is traversed during the *selection* phase, a code is selected by sampling the prior distribution [2]. In *Stochastic MuZero* each internal *decision* node is again expanded by querying the learned model, where

[2]In practice we follow the same quasi-random sampling approach as in [90] (A.3), where the code is selected using the formula $\arg\max_c \frac{Pr(c|as)}{N(c)+1}$.

the state of the *chance* parent node and a sampled code c are provided as an input, and the model returns a reward, a value and a policy. Similarly to *MuZero* the value of the newly added node is backpropagated up the tree, and the pUCT (2.6.2) formula is used to select an edge. The stochastic search used by *Stochastic MuZero* is shown schematically in figure 1.

7.2.3 Training

The stochastic model is unrolled and trained in an end-to-end fashion similar to *MuZero*. Specifically, given a trajectory of length K with observations $o_{\leq t:t+K}$, actions $a_{t:t+K}$, value targets $z_{t:t+K}$, policy targets $\pi_{t:t+K}$ and rewards $u_{t+1:t+K}$, the model is unrolled for K steps as shown in figure 1 and is trained to optimize the sum of two losses as shown in equation 7.4: a *MuZero* loss and a *chance* loss for learning the stochastic dynamics of the model.

$$L^{total} = L^{MuZero} + L^{chance} \tag{7.4}$$

The *MuZero* loss is the same as the one described in *MuZero* (see equation 6.3). The chance loss is applied to the predictions Q_t^k and σ_t^k which correspond to the latent *afterstates* as^k. The Q_t^k value is trained to match the value target z_{t+k} and the σ^k is trained towards the one-hot chance code $c_{t+k+1} = \text{one hot}(\arg\max_i(e(o^i_{\leq t+k+1})))$ produced by the encoder. Finally, following the standard VQ-VAE practice, we use a VQ-VAE commitment cost to ensure that the output of the encoder $c^e_{t+k} = e(o_{\leq t+k+1})$ is close to the code c_{t+k}. Equation 7.5 shows the chance loss used to train the model.

$$L_w^{chance} = \sum_{k=0}^{K-1} l^Q(z_{t+k}, Q_t^k) + \sum_{k=0}^{K-1} l^\sigma(c_{t+k+1}, \sigma_t^k) + \beta \underbrace{\sum_{k=0}^{K-1} \left\| c_{t+k+1} - c^e_{t+k+1} \right\|^2}_{\text{VQ-VAE commitment cost}} \tag{7.5}$$

7.3 Experiments

7.3.1 Results

We applied our algorithm to a variety of challenging stochastic and deterministic environments. First, we evaluated our approach in the classic game of *2048*, a

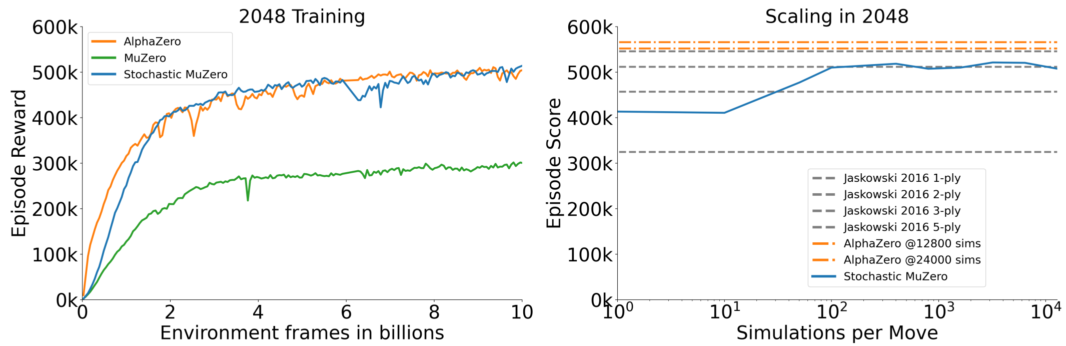


Figure 2: **Planning in *2048*.** a) *Stochastic MuZero*, trained using 100 simulations of planning with a learned stochastic model, matched the performance of *AlphaZero*, using 100 simulations of a perfect stochastic simulator, while a deterministic learned model (*MuZero*) performed poorly. b) Evaluation of final agent using different levels of search. *Stochastic MuZero* scales well during evaluation to intermediate levels of search (roughly comparable to 3-ply lookahead), exceeding the playing strength of the state-of-the-art baseline [63]. However, as the number of simulations increases we observe diminishing returns due to imperfections of the learned model.

stochastic single player game. Subsequently, we considered a two player zero-sum stochastic game, Backgammon, which belongs to the same class of board games such as Go, chess or Shogi where *MuZero* excels, but with stochasticity induced by the use of two dice. Finally, we evaluated our method in the deterministic game of Go, to measure any performance loss caused by the use of a stochastic model and search in deterministic environments in comparison to *MuZero*.

In each environment, we assess our algorithm's ability to learn a transition model and effectively use it during search. To this end, we compare *Stochastic MuZero* (using a stochastic learned model) to *MuZero* (using a deterministic learned model), *AlphaZero* (using a perfect simulator), and a strong baseline method (also using a perfect simulator). In the following sections we present our results for each environment separately.

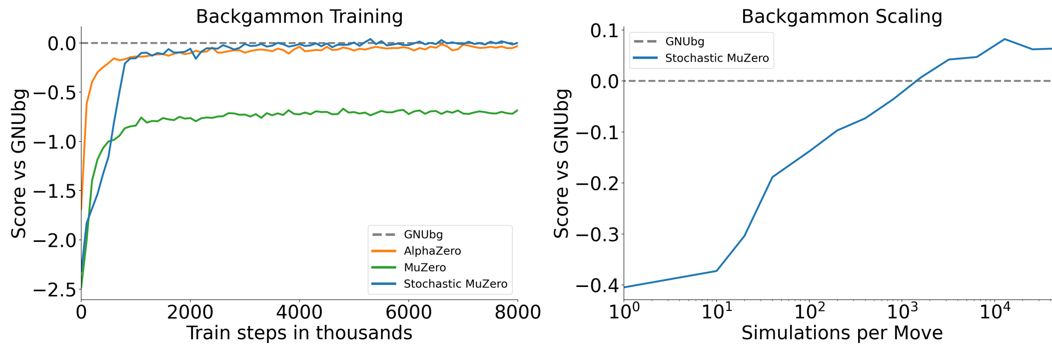


Figure 3: ***Stochastic MuZero* in Backgammon**. a) *Stochastic MuZero*, trained using 1600 simulations of planning with a learned stochastic model, matched the performance of *AlphaZero*, trained using 1600 simulations of a perfect stochastic simulator, as well as matching the superhuman-level program GNUbg Grandmaster. A deterministic learned model (*MuZero*) performed poorly. b) *Stochastic MuZero*'s model scaled well to large searches, and exceeded the playing strength of GNUbg Grandmaster when using more than 10^3 simulations.

2048

The game of *2048* (see 3.2.2) is a stochastic, single player, perfect information puzzle game played on a 4x4 board. Figure 2 compares the performance of *Stochastic MuZero* in *2048* to *AlphaZero*, *MuZero* and the state-of-the-art *Jaskowski 2016* agent [63]. Our method outperformed *Jaskowski 2016*, while using only a quarter of the training data. *Stochastic MuZero* also achieved the same performance as *AlphaZero* (using a perfect simulator), despite learning the model, and performed far better than *MuZero* (using a deterministic model).

Backgammon

Backgammon is a classic two player, zero-sum, stochastic board game (see 3.1.4); it was popularized as a standard testbed for reinforcement learning and artificial intelligence by TD-gammon [123]. Here we focus on the single game setting, where the final score takes the values ± 1 for a simple win or loss, ± 2 for a gammon and ± 3 for a backgammon.

In all experiments we compared to GNUbg Grandmaster [41], a superhuman-level open-source backgammon player. GNUbg combines a learned value function

based on handcrafted features with a specialized min-max tree search using a perfect stochastic simulator. GNUbg Grandmaster uses a 3-ply look-ahead search over a branching factor of 20 legal moves on average and 21 chance transitions.

Stochastic MuZero, using a learned stochastic model of the environment and only 1600 simulations per move, achieved the same playing strength as GNUbg, as shown in Figure 3b. The model learned by *Stochastic MuZero* is of high quality: it reached the same playing strength as *AlphaZero* (using a perfect stochastic simulator), and much higher strength than *MuZero* (using a deterministic learned model).

The model also robustly scaled to larger planning budgets (Figure 3c): the performance of *Stochastic MuZero* improved with increasing number of simulations per move, and ultimately exceeded the playing strength of GNUbg Grandmaster.

Given the high dimensionality of the action space in Backgammon, our Backgammon experiments used the sample-based search introduced by [59].

Go

Finally, we applied our method to the game of Go, since the goal of *Stochastic MuZero* is to extend the applicability of *MuZero* to stochastic environments while maintaining the latter's performance in deterministic environments. Figure 4 shows the Elo [26] achieved by *Stochastic MuZero* and *MuZero* during training. Although, *Stochastic MuZero* requires twice the number of network expansions in comparison to *MuZero* to achieve the same performance, due to the use of a stochastic MCTS instead of a deterministic one, we ensure that the methods are computationally equivalent by halving the network depth for the chance and dynamic parts of the *Stochastic MuZero*'s network.

7.3.2 Reproducibility

In order to evaluate the robustness of our method in all different environments, we replicated our experiments using three different initial random seeds. We observe that our method is robust to the random initialization and there is minimal variation in its performance between multiple runs. Due to the computational cost of each

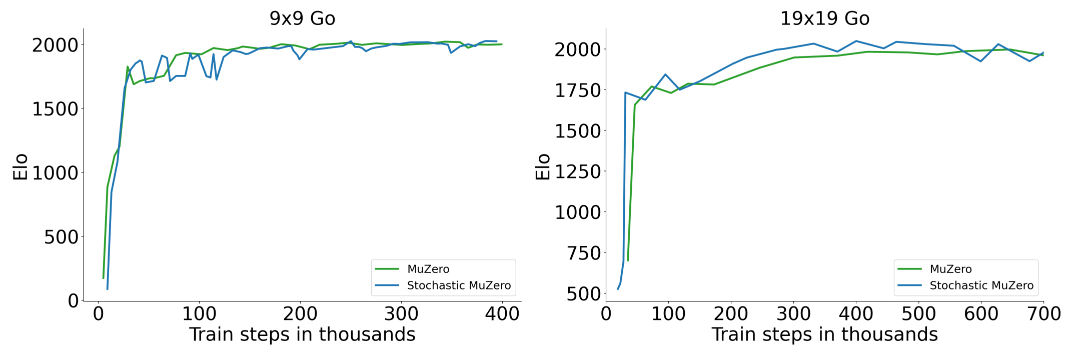


Figure 4: ***Stochastic MuZero* in Go**. Comparison of *Stochastic MuZero* and *MuZero* in the game of Go. a) *Stochastic MuZero* and *MuZero* when compared in 9x9 Go. *MuZero* has a search budget of 200 simulations during training of 800 during evaluation, while *Stochastic MuZero* uses 400 simulations during training and 1600 during evaluation. The Elo scale was anchored so that the performance of the final *MuZero* baseline corresponded to an Elo of 2000. b) *Stochastic MuZero* and *MuZero* when compared in 19x19 Go. *MuZero* has a search budget of 400 simulations during training of 800 during evaluation, while *Stochastic MuZero* uses 800 simulations during training and 1600 during evaluation. The Elo scale was anchored so that the performance of the final *MuZero* baseline corresponded to an Elo of 2000.

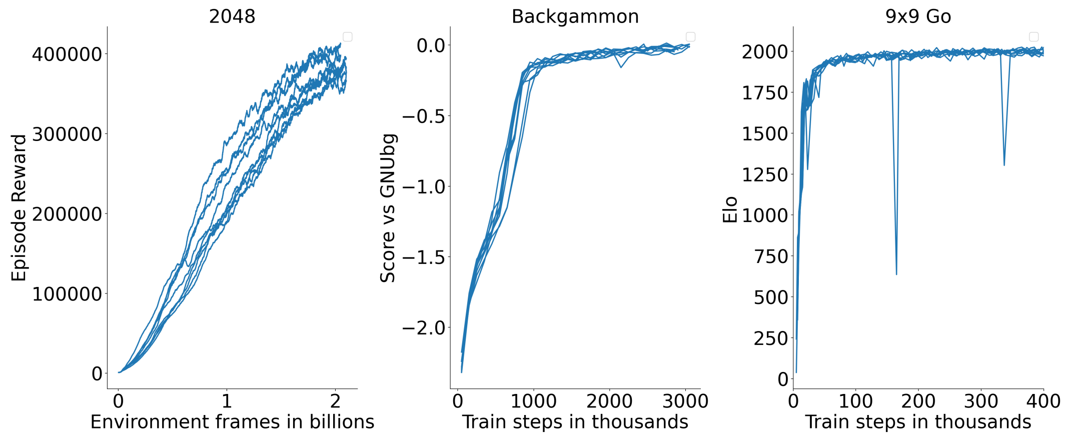


Figure 5: ***Stochastic MuZero* reproducibility across all domains.** We ran our method *Stochastic MuZero* in all environments using 9 different seeds to measure its robustness to random initialization. We observed that there is minimal variation in the performance of *Stochastic MuZero* for all different seeds. Due to the computational cost of each experiment we used a smaller number of training steps for each experiment.

experiment we used a smaller number of training steps for each experiment.

7.3.3 Ablations

In order to investigate the distribution of chance outcomes at each chance node for *Stochastic MuZero*, we collected a dataset for each game by storing the probability distribution over chance nodes, $\sigma_t^k = Pr(c_{t+k+1}|as_t^k)$, for all afterstate prediction network evaluations invoked throughout all searches in 5 episodes. Subsequently, we sorted each chance node distribution and finally, we computed the average distribution, as shown in Figure 6. We observed that in the case of a deterministic environment like Go, the chance distribution collapsed to a single code, while in stochastic environments the model used multiple codes. Furthermore, in Backgammon, the chance distribution had a support of 21 codes with non-negligible probability, which corresponds to the number of distinct rolls of two dice.

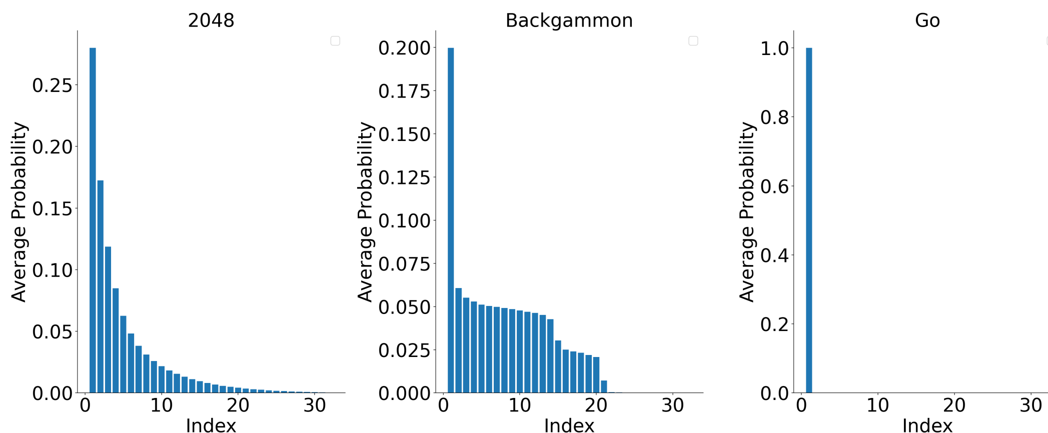


Figure 6: **Average distribution of learned chance outcomes.** The average distribution of learned chance outcomes over all chance nodes after running *Stochastic MuZero* at each game for 5 episodes.

7.4 Conclusions

In this chapter, we presented a new method for learning a stochastic model of the environment, in a fully online reinforcement learning setting, and showed that the learned model can be effectively combined with planning. Our approach builds on top of *MuZero*, a model-based reinforcement learning agent that has been widely successful in a range of environments and settings, but its applicability is limited to deterministic or weakly stochastic environments.

We have shown that our algorithm, *Stochastic MuZero*, can overcome the limitations of *MuZero*, significantly outperforming it in stochastic environments, and achieving the same or better performance than an equivalent program (*AlphaZero*) that makes use of a perfect simulator for the environment. Finally, we have demonstrated that *Stochastic MuZero* matched or exceeded the performance of previous methods that use a perfect stochastic simulator, in a pure reinforcement learning setting without using any prior knowledge about the environment.

Chapter 8

Conclusions

8.1 Open Problems in Learning and Planning

Model-based reinforcement learning has traditionally focused on the development of methods that combine models of the environment dynamics with some form of planning. The learned model usually operates at the time granularity of single time steps, while the planning method assumes that the model can perfectly emulate the dynamics of the real environment. However, this is different from biological systems, which tend to construct plans at abstract time scales and assume that their future predictions can be inaccurate.

Planning at the level of single actions can limit the applicability of classic model-based methods in domains where the agent needs to make decisions at different time scales (i.e. robotics). Applying our powerful search algorithms to these problems requires the development of solutions which employ temporally abstracted models that can make predictions at different time scales. Designing such models and learning abstraction spaces which can best explain the environment dynamics is a challenging problem left for future research.

Recent developments in the field of deep learning have provided the tools to obtain really powerful models of the transition dynamics of a wide range of environments. However, these models still suffer from approximation errors when employed in really complex domains or when they need to make predictions about events far into the future. This can be catastrophic when they are combined with

search methods, such as MCTS, which assume that the model is perfect. Our *MuZero* (see chapter 6) experiments have demonstrated these problems in an experimental setting, with diminishing returns for an increased search budget in domains such as Atari or with the difference in performance between *AlphaZero* and *MuZero* for extremely long searches in the game of Go. A promising future research direction is to couple the search and learning into a unified framework so that the planning module is trained to account for errors in the model predictions.

8.2 The promise of Learning and Planning

In this thesis we presented a series of general purpose reinforcement learning methods that rely on planning to achieve superhuman performance in a wide range of challenging domains. We have developed *AlphaZero* (see chapter 5), a general reinforcement learning algorithm which combines a tree based search with deep neural networks and is trained completely from scratch without any domain specific adaptations, only through games of self-play. *AlphaZero* achieved superhuman performance in the games of Go and chess, domains which have historically acted as standard testbeds for the development of new ideas in the field of artificial intelligence. Subsequently, we applied our powerful tree based search method to the visually complex and challenging domain of Atari, by employing a learned model of the environment dynamics. The resulting algorithm, *MuZero* (see chapter 6) outperformed all previous approaches. It overcomes the limitations of previous model-based approaches by training its model to predict only those quantities that matter for planning. Finally, we extended the applicability of our approach to highly stochastic problems. Our algorithm, *Stochastic MuZero* (see chapter 7), adapts *MuZero*'s tree search to account for stochastic transitions and introduces a novel training regime for learning a stochastic model of the environment dynamics. *Stochastic MuZero* achieved superhuman performance in challenging stochastic domains such as backgammon and *2048*, while performing on par with *MuZero* in Go.

Constructing plans and executing them is an important feature of human and animal behaviour. Research in human psychology [64] has shown that humans

construct mental models of the world and use them for planning and inference. As we strive to develop more sophisticated artificial agents it is important to design methods which can plan and construct models of their environment.

As Richard Sutton said in his bitter lesson [115], the past few decades of AI research have shown that the most successful methods in tackling hard problems are the ones which can scale efficiently with more data and compute. Approaches like *MuZero* that combine planning and learning are well positioned to take advantage of the increasing availability of computational resources and data. Devising better such techniques may provide unprecedented dividends in the future.

Appendix A

AlphaZero Appendix

A.1 Domain knowledge in *AlphaZero*

AlphaZero was provided with the following domain knowledge about each game:

1. The input features describing the position, and the output features describing the move, are structured as a set of planes; i.e. the neural network architecture is matched to the grid-structure of the board.

2. *AlphaZero* is provided with perfect knowledge of the game rules. These are used during MCTS, to simulate the positions resulting from a sequence of moves, to determine game termination, and to score any simulations that reach a terminal state.

3. Knowledge of the rules is also used to encode the input planes (i.e. castling, repetition, no-progress) and output planes (how pieces move, promotions, and piece drops in shogi).

4. The typical number of legal moves is used to scale the exploration noise (see below).

5. Chess and shogi games exceeding 512 steps were terminated and assigned a drawn outcome; Go games exceeding 722 steps were terminated and scored with Tromp-Taylor rules.

AlphaZero did not use an opening book, endgame tablebases, or domain-specific heuristics.

A.2 Experimental Setup

A.2.1 Network Input Representation

We describe the representation of the board inputs, and the representation of the action outputs, used by the neural network in *AlphaZero*. Other representations could have been used; in our experiments the training algorithm worked robustly for many reasonable choices.

The input to the neural network is an $N \times N \times (MT + L)$ image stack that represents state using a concatenation of T sets of M planes of size $N \times N$. Each set of planes represents the board position at a time-step $t - T + 1, ..., t$, and is set to zero for time-steps less than 1. The board is oriented to the perspective of the current player. The M feature planes are composed of binary feature planes indicating the presence of the player's pieces, with one plane for each piece type, and a second set of planes indicating the presence of the opponent's pieces. For shogi there are additional planes indicating the number of captured prisoners of each type. There are an additional L constant-valued input planes denoting the player's colour, the move number, and the state of special rules: the legality of castling in chess (kingside or queenside); the repetition count for the current position (3 repetitions is an automatic draw in chess; 4 in shogi); and the number of moves without progress in chess (50 moves without progress is an automatic draw). Input features are summarized in Table 1.

A move in chess may be described in two parts: first selecting the piece to move, and then selecting among possible moves for that piece. We represent the policy $\pi(a|s)$ by a $8 \times 8 \times 73$ stack of planes encoding a probability distribution over 4,672 possible moves. Each of the 8×8 positions identifies the square from which to "pick up" a piece. The first 56 planes encode possible 'queen moves' for any piece: a number of squares $[1..7]$ in which the piece will be moved, along one of eight relative compass directions $\{N, NE, E, SE, S, SW, W, NW\}$. The next 8 planes encode possible knight moves for that piece. The final 9 planes encode possible underpromotions for pawn moves or captures in two possible diagonals, to knight, bishop or rook respectively. Other pawn moves or captures from the seventh rank

Go		Chess		Shogi	
Feature	Planes	Feature	Planes	Feature	Planes
P1 stone	1	P1 piece	6	P1 piece	14
P2 stone	1	P2 piece	6	P2 piece	14
		Repetitions	2	Repetitions	3
				P1 prisoner count	7
				P2 prisoner count	7
Colour	1	Colour	1	Colour	1
		Total move count	1	Total move count	1
		P1 castling	2		
		P2 castling	2		
		No-progress count	1		
Total	17	Total	119	Total	362

Table 1: Input features used by *AlphaZero* in Go, chess and shogi respectively. The first set of features are repeated for each position in a $T = 8$-step history. Counts are represented by a single real-valued input; other input features are represented by a one-hot encoding using the specified number of binary input planes. The current player is denoted by P1 and the opponent by P2.

Chess		Shogi	
Feature	Planes	Feature	Planes
Queen moves	56	Queen moves	64
Knight moves	8	Knight moves	2
Underpromotions	9	Promoting queen moves	64
		Promoting knight moves	2
		Drop	7
Total	73	Total	139

Table 2: Action representation used by *AlphaZero* in chess and shogi respectively. The policy is represented by a stack of planes encoding a probability distribution over legal moves; planes correspond to the entries in the table.

are promoted to a queen.

The policy in shogi is represented by a $9 \times 9 \times 139$ stack of planes similarly encoding a probability distribution over 11,259 possible moves. The first 64 planes encode 'queen moves' and the next 2 planes encode knight moves. An additional $64+2$ planes encode promoting queen moves and promoting knight moves respectively. The last 7 planes encode a captured piece dropped back into the board at that location.

The policy in Go is represented using a flat distribution over $19 \times 19 + 1$ moves representing possible stone placements and the pass move. We also tried using a flat distribution over moves for chess and shogi; the final result was almost identical although training was slightly slower.

Illegal moves are masked out by setting their probabilities to zero, and re-normalising the probabilities over the remaining set of legal moves.

The action representations are summarized in Table 2.

A.2.2 Network Architecture

The neural network consists of a "body" followed by both policy and value "heads". The body consists of a rectified batch-normalized convolutional layer followed by

19 residual blocks [50]. Each such block consists of two rectified batch-normalized convolutional layers with a skip connection. Each convolution applies 256 filters of kernel size 3×3 with stride 1. The policy head applies an additional rectified, batch-normalized convolutional layer, followed by a final convolution of 73 filters for chess or 139 filters for shogi, or a linear layer of size 362 for Go, representing the logits of the respective policies described above. The value head applies an additional rectified, batch-normalized convolution of 1 filter of kernel size 1×1 with stride 1, followed by a rectified linear layer of size 256 and a tanh-linear layer of size 1.

A.2.3 Configuration

During training, each MCTS used 800 simulations. The number of games, positions, and thinking time varied per game due largely to different board sizes and game lengths, and are shown in Table 5.3. The learning rate was set to 0.2 for each game, and was dropped three times during the course of training to 0.02, 0.002 and 0.0002 respectively, after 100, 300 and 500 thousands of steps for chess and shogi, and after 0, 300 and 500 thousands of steps for Go. Moves are selected in proportion to the root visit count. Dirichlet noise Dir(α) was added to the prior probabilities in the root node; this was scaled in inverse proportion to the approximate number of legal moves in a typical position, to a value of $\alpha = \{0.3, 0.15, 0.03\}$ for chess, shogi and Go respectively. Positions were batched across parallel training games for evaluation by the neural network.

During evaluation, *AlphaZero* selects moves greedily with respect to the root visit count. Each MCTS was executed on a single machine with 4 first-generation TPUs.

A.2.4 Opponents

To evaluate performance in chess, we used *Stockfish* version 8 (official Linux release) as a baseline program. *Stockfish* was configured according to its 2016 TCEC world championship superfinal settings: 44 threads on 44 cores (two 2.2GHz Intel Xeon Broadwell CPUs with 22 cores), a hash size of 32GB, syzygy endgame

tablebases, at 3 hour time controls with 15 additional seconds per move. We also evaluated against the most recent version, *Stockfish* 9 (just released at time of writing), using the same configuration.

Stockfish does not have an opening book of its own and all primary evaluations were performed without an opening book. We also performed one secondary evaluation in which the opponent's opening moves were selected by the *Brainfish* program, using an opening book derived from *Stockfish*. However, we note that these matches were low in diversity, and *AlphaZero* and *Stockfish* tended to produce very similar games throughout the match, more than 90% of which were draws. When we forced *AlphaZero* to play with greater diversity (by softmax sampling with a temperature of 10.0 among moves for which the value was no more than 1% away from the best move for the first 30 plies) the winning rate increased from 5.8% to 14%.

To evaluate performance in shogi, we used *Elmo* version WCSC27 in combination with YaneuraOu 2017 Early KPPT 4.79 64AVX2 TOURNAMENT as a baseline program, using 44 CPU threads (on two 2.2GHz Intel Xeon Broadwell CPUs with 22 cores) and a hash size of 32GB with the usi options of EnteringKingRule set to CSARule27, MinimumThinkingTime set to 1000, BookFile set to standard_book.db, BookDepthLimit set to 0 and BookMoves set to 200. Additionally, we also evaluated against *Aperyqhapaq* combined with the same YaneuraOu version and no book file. For *Aperyqhapaq*, we used the same usi options as for *Elmo* except for the book setting.

A.2.5 Match conditions

We measured the head-to-head performance of *AlphaZero* in matches against each of the above opponents (Figure 5.8). Three types of match were played: starting from the initial board position (the default configuration, unless otherwise specified); starting from human opening positions; or starting from the 2016 TCEC opening positions [1].

[1] The TCEC world championship disallows opening books and instead starts two games (one from each colour) from each opening position.

The majority of matches for chess, shogi and Go used the 2016 TCEC superfinal time controls: 3 hours of main thinking time, plus 15 additional seconds of thinking time for each move. We also investigated asymmetric time controls (Figure 5.8B), where the opponent received 3 hours of main thinking time but *AlphaZero* received only a fraction of this time. Finally, for shogi only, we ran a match using faster time controls used in the 2017 CSA world championship: 10 minutes per game plus 10 seconds per move.

AlphaZero used a simple time control strategy: thinking for $1/20$th of the remaining time. Opponent programs used customized, sophisticated heuristics for time control. Pondering was disabled for all players (particularly important for the asymmetric time controls in Figure 5.8).

Resignation was enabled for all players (-650 centipawns for 4 consecutive moves for *Stockfish*, -4,500 centipawns for 10 consecutive moves for *Elmo*, or a value of -0.9 for *AlphaZero* and *AlphaGo Lee*).

Matches consisted of 1,000 games, except for the human openings (200 games as black and 200 games as white from each opening) and the 2016 TCEC openings (50 games as black and 50 games as white from each of the 50 openings). The human opening positions were chosen as those played more than 100,000 times in an online database [2].

A.2.6 Elo ratings

We evaluated the relative strength of *AlphaZero* (Figure 5.7) by measuring the Elo rating of each player. We estimate the probability that player a will defeat player b by a logistic function $p(a \text{ defeats } b) = (1 + 10^{(c_{\text{elo}}(e(b)-e(a)))})^{-1}$, and estimate the ratings $e(\cdot)$ by Bayesian logistic regression, computed by the *BayesElo* program [26] using the standard constant $c_{\text{elo}} = 1/400$.

Elo ratings were computed from the results of a 1 second per move tournament between iterations of *AlphaZero* during training, and also a baseline player: either *Stockfish*, *Elmo* or *AlphaGo Lee* respectively. The Elo rating of the baseline players was anchored to publicly available values.

In order to compare Elo ratings at 1 second per move time controls to standard

Program	Win	Draw	Loss
Stockfish	57.1 %	42.9 %	0.0 %
Elmo	98.7 %	1.0 %	0.3 %

Table 3: Performance comparison of *Stockfish* and *Elmo*, when using full time controls of 3h per game, compared to time controls of 1s per move.

Elo ratings at full time controls, we also provide the results of *Stockfish* vs. *Stockfish* and *Elmo* vs. *Elmo* matches (Table 3).

A.3 Chess Openings

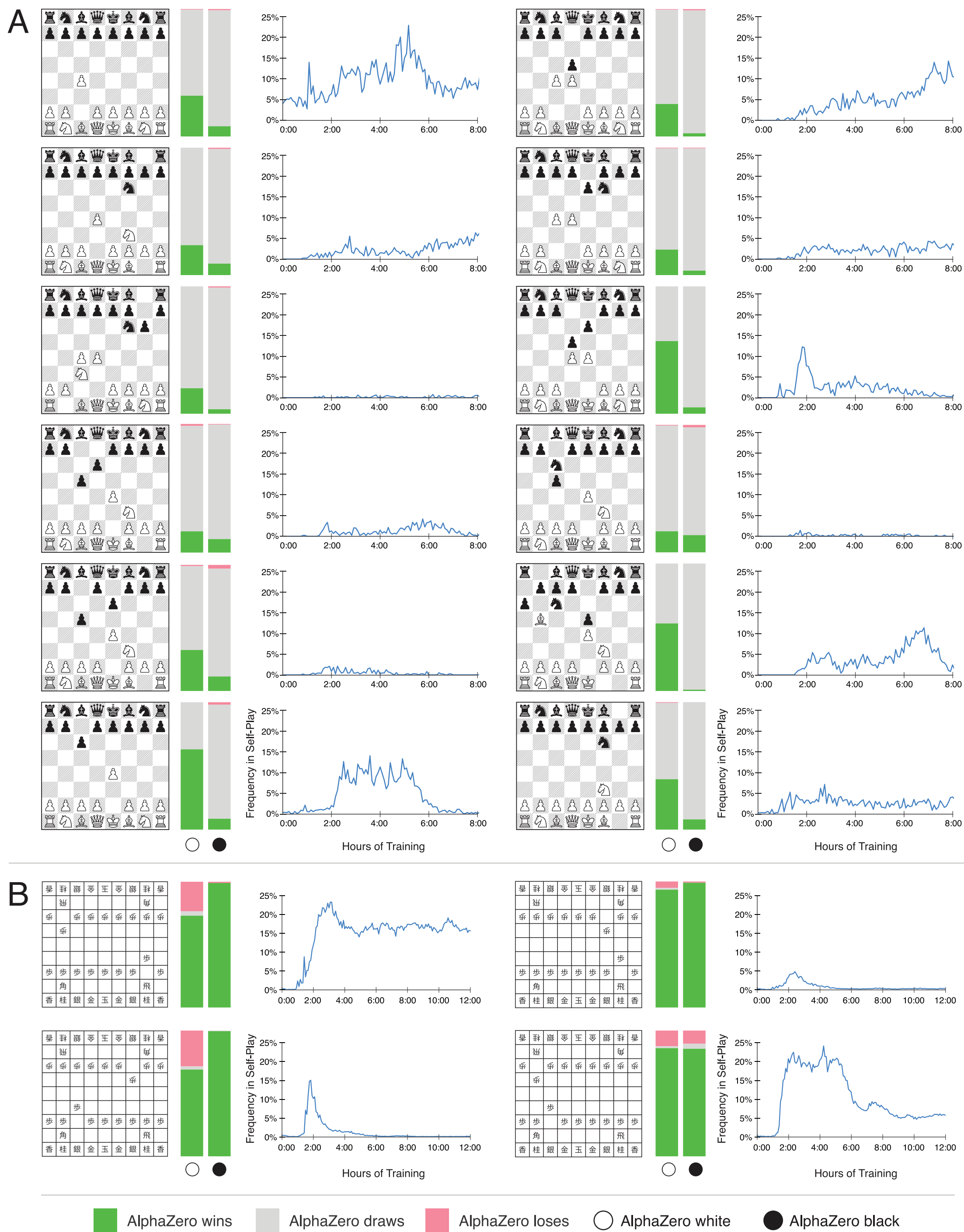


Figure 1: **Matches starting from the most popular human openings**. *AlphaZero* plays against **(A)** *Stockfish* in chess and **(B)** *Elmo* in shogi. In the left bar, *AlphaZero* plays white, starting from the given position; in the right bar *AlphaZero* plays black. Each bar shows the results from *AlphaZero*'s perspective: win (green), draw (grey), loss (red). The percentage frequency of self-play training games in which this opening was selected by *AlphaZero* is plotted against the duration of training, in hours.

Appendix B

MuZero Appendix

B.1 Hyperparameters

For simplicity we preferentially use the same architectural choices and hyperparameters as in previous experiments. Specifically, we started with the network architecture and search choices of *AlphaZero*. For board games, we use the same UCB constants, dirichlet exploration noise and the same 800 simulations per search as in *AlphaZero*.

Due to the much smaller branching factor and simpler policies in Atari, we only used 50 simulations per search to speed up experiments. As shown in Figure 3B, the algorithm is not very sensitive to this choice. We also use the same discount (0.997) and value transformation (see Network Architecture section) as R2D2 [68].

For parameter values not mentioned in the text, please refer to the pseudocode.

B.2 Data Generation

To generate training data, the latest checkpoint of the network (updated every 1000 training steps) is used to play games with MCTS. In the board games Go, chess and shogi the search is run for 800 simulations per move to pick an action; in Atari due to the much smaller action space 50 simulations per move are sufficient.

For board games, games are sent to the training job as soon as they finish. Due to the much larger length of Atari games (up to 30 minutes or 108,000 frames), intermediate sequences are sent every 200 moves. In board games, the training job keeps an in-memory replay buffer of the most recent 1 million games received;

in Atari, where the visual observations are larger, the most recent 125 thousand sequences of length 200 are kept.

During the generation of experience in the board game domains, the same exploration scheme as the one described in *AlphaZero* is used. Using a variation of this scheme, in the Atari domain actions are sampled from the visit count distribution throughout the duration of each game, instead of just the first k moves. Moreover, the visit count distribution is parametrized using a temperature parameter T:

$$p_\alpha = \frac{N(\alpha)^{1/T}}{\sum_b N(b)^{1/T}} \tag{B.1}$$

T is decayed as a function of the number of training steps of the network. Specifically, for the first 500k training steps a temperature of 1 is used, for the next 250k steps a temperature of 0.5 and for the remaining 250k a temperature of 0.25. This ensures that the action selection becomes greedier as training progresses.

B.3 Network Input

B.3.1 Representation Function

The history over board states was used as input to the representation function for Go, chess and shogi is represented similarly to *AlphaZero*. In Go and shogi we encode the last 8 board states as in *AlphaZero*; in chess we increased the history to the last 100 board states to allow correct prediction of draws.

For Atari, the input of the representation function includes the last 32 RGB frames at resolution 96x96 along with the last 32 actions that led to each of those frames. We encode the historical actions because unlike board games, an action in Atari does not necessarily have a visible effect on the observation. RGB frames are encoded as one plane per color, rescaled to the range $[0,1]$, for red, green and blue respectively. We perform no other normalization, whitening or other preprocessing of the RGB input. Historical actions are encoded as simple bias planes, scaled as $a/18$ (there are 18 total actions in Atari).

B.3.2 Dynamics Function

The input to the dynamics function is the hidden state produced by the representation function or previous application of the dynamics function, concatenated with a representation of the action for the transition. Actions are encoded spatially in planes of the same resolution as the hidden state. In Atari, this resolution is 6x6 (see description of downsampling in Network Architecture section), in board games this is the same as the board size (19x19 for Go, 8x8 for chess, 9x9 for shogi).

In Go, a normal action (playing a stone on the board) is encoded as an all zero plane, with a single one in the position of the played stone. A pass is encoded as an all zero plane.

In chess, 8 planes are used to encode the action. The first one-hot plane encodes which position the piece was moved from. The next two planes encode which position the piece was moved to: a one-hot plane to encode the target position, if on the board, and a second binary plane to indicate whether the target was valid (on the board) or not. This is necessary because for simplicity our policy action space enumerates a superset of all possible actions, not all of which are legal, and we use the same action space for policy prediction and to encode the dynamics function input. The remaining five binary planes are used to indicate the type of promotion, if any (queen, knight, bishop, rook, none).

The encoding for shogi is similar, with a total of 11 planes. We use the first 8 planes to indicate where the piece moved from - either a board position (first one-hot plane) or the drop of one of the seven types of prisoner (remaining 7 binary planes). The next two planes are used to encode the target as in chess. The remaining binary plane indicates whether the move was a promotion or not.

In Atari, an action is encoded as a one hot vector which is tiled appropriately into planes.

B.4 Network Architecture

The prediction function $p^k, v^k = f_\theta(s^k)$ uses the same architecture as *AlphaZero*: one or two convolutional layers that preserve the resolution but reduce the number

of planes, followed by a fully connected layer to the size of the output.

For value and reward prediction in Atari we follow [95] in scaling targets using an invertible transform $h(x) = sign(x)(\sqrt{|x|+1} - 1) + \varepsilon x$, where $\varepsilon = 0.001$ in all our experiments. We then apply a transformation ϕ to the scalar reward and value targets in order to obtain equivalent categorical representations. We use a discrete support set of size 601 with one support for every integer between -300 and 300. Under this transformation, each scalar is represented as the linear combination of its two adjacent supports, such that the original value can be recovered by $x = x_{low} * p_{low} + x_{high} * p_{high}$. As an example, a target of 3.7 would be represented as a weight of 0.3 on the support for 3 and a weight of 0.7 on the support for 4. The value and reward outputs of the network are also modeled using a softmax output of size 601. During inference the actual value and rewards are obtained by first computing their expected value under their respective softmax distribution and subsequently by inverting the scaling transformation. Scaling and transformation of the value and reward happens transparently on the network side and is not visible to the rest of the algorithm.

Both the representation and dynamics function use the same architecture as *AlphaZero*, but with 16 instead of 20 residual blocks [50]. We use 3x3 kernels and 256 hidden planes for each convolution.

For Atari, where observations have large spatial resolution, the representation function starts with a sequence of convolutions with stride 2 to reduce the spatial resolution. Specifically, starting with an input observation of resolution 96x96 and 128 planes (32 history frames of 3 color channels each, concatenated with the corresponding 32 actions broadcast to planes), we downsample as follows:

- 1 convolution with stride 2 and 128 output planes, output resolution 48x48.
- 2 residual blocks with 128 planes
- 1 convolution with stride 2 and 256 output planes, output resolution 24x24.
- 3 residual blocks with 256 planes.
- Average pooling with stride 2, output resolution 12x12.

- 3 residual blocks with 256 planes.
- Average pooling with stride 2, output resolution 6x6.

The kernel size is 3x3 for all operations.

For the dynamics function (which always operates at the downsampled resolution of 6x6), the action is first encoded as an image, then stacked with the hidden state of the previous step along the plane dimension.

B.5 Training

During training, the *MuZero* network is unrolled for K hypothetical steps and aligned to sequences sampled from the trajectories generated by the MCTS actors. Sequences are selected by sampling a state from any game in the replay buffer, then unrolling for K steps from that state. In Atari, samples are drawn according to prioritized replay [101], with priority $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$, where $p_i = |\nu_i - z_i|$, ν is the search value and z the observed n-step return. To correct for sampling bias introduced by the prioritized sampling, we scale the loss using the importance sampling ratio $w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^\beta$. In all our experiments, we set $\alpha = \beta = 1$. For board games, states are sampled uniformly.

Each observation o_t along the sequence also has a corresponding search policy π_t, search value function, ν_t and environment reward u_t. At each unrolled step k the network has a loss to the policy, value and reward target for that step, summed to produce the total loss for the *MuZero* network (see Equation 6.3). Note that, in board games without intermediate rewards, we omit the reward prediction loss. For board games, we bootstrap directly to the end of the game, equivalent to predicting the final outcome; for Atari we bootstrap for $n = 10$ steps into the future.

To maintain roughly similar magnitude of gradient across different unroll steps, we scale the gradient in two separate locations:

- We scale the loss of each head by $\frac{1}{K}$, where K is the number of unroll steps. This ensures that the total gradient has similar magnitude irrespective of how many steps we unroll for.

- We also scale the gradient at the start of the dynamics function by $\frac{1}{2}$. This ensures that the total gradient applied to the dynamics function stays constant.

In the experiments reported in this paper, we always unroll for $K = 5$ steps. For a detailed illustration, see Figure 1.

To improve the learning process and bound the activations, we also scale the hidden state to the same range as the action input ($[0,1]$): $s_{scaled} = \frac{s - \min(s)}{\max(s) - \min(s)}$.

All experiments were run using third generation Google Cloud TPUs [43]. For each board game, we used 16 TPUs for training and 1000 TPUs for selfplay. For each game in Atari, we used 8 TPUs for training and 32 TPUs for selfplay. The much smaller proportion of TPUs used for acting in Atari is due to the smaller number of simulations per move (50 instead of 800) and the smaller size of the dynamics function compared to the representation function.

Note that the network is trained separately for each environment (i.e. one model for each different Atari game or board game). However, in principle the same model could be shared between different environments during training, or could be tested in new environments (i.e. zero-shot generalisation); this approach is left to future work.

B.6 *MuZero* Equations

Model

$$\left.\begin{array}{ll} s^0 & = h_\theta(o_1, ..., o_t) \\ r^k, s^k & = g_\theta(s^{k-1}, a^k) \\ p^k, v^k & = f_\theta(s^k) \end{array}\right\} \; p^k, v^k, r^k = \mu_\theta(o_1, ..., o_t, a^1, ..., a^k)$$

Search

$$\nu_t, \pi_t = MCTS(s_t^0, \mu_\theta)$$

$$a_t \sim \pi_t$$

Learning Rule

$$p_t^k, v_t^k, r_t^k = \mu_\theta(o_1, ..., o_t, a_{t+1}, ..., a_{t+k})$$

$$z_t = \begin{cases} u_T & \text{for games} \\ u_{t+1} + \gamma u_{t+2} + ... + \gamma^{n-1} u_{t+n} + \gamma^n \nu_{t+n} & \text{for general MDPs} \end{cases}$$

$$l_t(\theta) = \sum_{k=0}^{K} l^p(\pi_{t+k}, p_t^k) + \sum_{k=0}^{K} l^v(z_{t+k}, v_t^k) + \sum_{k=1}^{K} l^r(u_{t+k}, r_t^k) + c||\theta||^2$$

Losses

$$l^p(\pi, p) = \pi^T \log p$$

$$l^v(z, v) = \begin{cases} (z - v)^2 & \text{for games} \\ \phi(z)^T \log v & \text{for general MDPs} \end{cases}$$

$$l^r(u, r) = \begin{cases} 0 & \text{for games} \\ \phi(u)^T \log r & \text{for general MDPs} \end{cases}$$

Figure 1: **Equations summarising the *MuZero* algorithm**. Here, $\phi(x)$ refers to the representation of a real number x through a linear combination of its adjacent integers, as described in the Network Architecture section.

Bibliography

[1] Kgs games database, 2017. URL: https://u-go.net/gamerecords/.

[2] Online chess games database, 365chess, 2017. URL: https://www.365chess.com/.

[3] Ioannis Antonoglou, Julian Schrittwieser, Sherjil Ozair, Thomas K Hubert, and David Silver. Planning in stochastic environments with a learned model. In *International Conference on Learning Representations*, 2022.

[4] Oleg Arenz. Monte carlo chess. bachelor thesis, Technische Universität Darmstadt, Darmstadt, 2022.

[5] Computer Shogi Association. Results of the 27th world computer shogi championship. http://www2.computer-shogi.org/wcsc27/index_e.html. Retrieved November 29th, 2017.

[6] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2–3):235–256, may 2002.

[7] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM J. Comput.*, 32(1):48–77, jan 2003.

[8] Kamyar Azizzadenesheli, Brandon Yang, Weitang Liu, Emma Brunskill, Zachary C. Lipton, and Animashree Anandkumar. Surprising negative results for generative adversarial tree search. *CoRR*, abs/1806.05780, 2018.

[9] Petr Baudiš and Jean-loup Gailly. Pachi: State of the art open source go program. In H. Jaap van den Herik and Aske Plaat, editors, *Advances in Computer Games*, pages 24–38, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[10] J. Baxter, A. Tridgell, and L. Weaver. Learning to play chess using temporal differences. *Machine Learning*, 40(3):243–263, 2000.

[11] Donald F. Beal and Martin C. Smith. Temporal difference learning for heuristic search and game playing. *Inf. Sci.*, 122(1):3–21, 2000.

[12] Donald F. Beal and Martin C. Smith. Temporal difference learning applied to game playing and the results of application to shogi. *Theoretical Computer Science*, 252(1–2):105–119, 2001.

[13] Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.

[14] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *ICML*, 2017.

[15] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[16] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[17] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[18] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry,

Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[19] Michael Buro. From simple features to sophisticated evaluation functions. In H. Jaap van den Herik and Hiroyuki Iida, editors, *Computers and Games*, pages 126–145, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[20] M. Campbell, A. J. Hoane, and F. Hsu. Deep Blue. *Artificial Intelligence*, 134:57–83, 2002.

[21] Tristan Cazenave. Residual networks for computer go. *IEEE Transactions on Games*, 10(1):107–110, 2018.

[22] Silvia Chiappa, Sébastien Racanière, Daan Wierstra, and Shakir Mohamed. Recurrent environment simulators. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[23] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, Doha, Qatar, October 2014. Association for Computational Linguistics.

[24] Christopher Clark and Amos Storkey. Training deep convolutional neural networks to play Go. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of

Proceedings of Machine Learning Research, pages 1766–1774, Lille, France, 07–09 Jul 2015. PMLR.

[25] Adrien Couetoux. *Monte Carlo Tree Search for Continuous and Stochastic Sequential Decision Making Problems*. Thesis, Université Paris Sud - Paris XI, September 2013.

[26] R. Coulom. Whole-history rating: A Bayesian rating system for players of time-varying strength. In *International Conference on Computers and Games*, pages 113–124, 2008.

[27] Rémi Coulom. Computing ”Elo ratings” of move patterns in the game of Go. *J. Int. Comput. Games Assoc.*, 30:198–208, 2007.

[28] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit quantile networks for distributional reinforcement learning. In *ICML*, 2018.

[29] Will Dabney, Mark Rowland, Marc G. Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression. *CoRR*, abs/1710.10044, 2018.

[30] Mogens Dalgaard, Felix Motzoi, Jens Jakob W. H. Sørensen, and Jacob Friis Sherson. Global optimization of quantum dynamics with AlphaZero deep exploration. *npj Quantum Information*, 6:1–9, 2019.

[31] Ivo Danihelka, Arthur Guez, Julian Schrittwieser, and David Silver. Policy improvement by planning with Gumbel. In *International Conference on Learning Representations*, 2022.

[32] Omid E David, Nathan S Netanyahu, and Lior Wolf. Deepchess: End-to-end deep neural network for automatic learning in chess. In *International Conference on Artificial Neural Networks*, pages 88–96. Springer, 2016.

[33] M. Enzenberger. *Evaluation in Go by a Neural Network Using Soft Segmentation*, pages 97–108. Springer US, Boston, MA, 2004.

[34] Markus Enzenberger. The integration of a priori knowledge into a Go playing neural network. 1996.

[35] Markus Enzenberger, Martin Müller, Broderick Arneson, and Richard Segal. Fuego—An Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010.

[36] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning*, pages 1407–1416. PMLR, 2018.

[37] Amir-massoud Farahmand. Iterative value-aware model learning. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 9090–9101, Red Hook, NY, USA, 2018. Curran Associates Inc.

[38] Amir-Massoud Farahmand, Andre Barreto, and Daniel Nikovski. Value-Aware Loss Function for Model-based Reinforcement Learning. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 1486–1494, Fort Lauderdale, FL, USA, 20–22 Apr 2017. PMLR.

[39] Gregory Farquhar, Tim Rocktäschel, Maximilian Igl, and Shimon Whiteson. TreeQN and AtreeC: Differentiable tree planning for deep reinforcement learning. 10 2017.

[40] Inc. Free Software Foundation. Gnu chess.

[41] Inc. Free Software Foundation. Gnu backgammon, 2004.

[42] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202, 1980.

[43] Cloud TPU. `https://cloud.google.com/tpu/`.

[44] David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[45] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2555–2565. PMLR, 09–15 Jun 2019.

[46] Danijar Hafner, Timothy P Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models. In *International Conference on Learning Representations*, 2021.

[47] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[48] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction.* Springer, 2 edition, 2009.

[49] Frederick Hayes-Roth, Donald A Waterman, and Douglas B Lenat. *Building expert systems.* Addison-Wesley Longman Publishing Co., Inc., 1983.

[50] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *14th European Conference on Computer Vision*, pages 630–645, 2016.

[51] Mikael Henaff, William F Whitney, and Yann LeCun. Model-based planning with discrete and continuous actions. *arXiv preprint arXiv:1705.07177*, 2017.

[52] Matteo Hessel, Ivo Danihelka, Fabio Viola, Arthur Guez, Simon Schmitt, Laurent Sifre, Theophane Weber, David Silver, and Hado Van Hasselt. Muesli: Combining improvements in policy optimization. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 4214–4226. PMLR, 18–24 Jul 2021.

[53] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*, 2017.

[54] Kunihito Hoki and Tomoyuki Kaneko. Large-scale optimization for evaluation functions with minimax search. *Journal of Artificial Intelligence Research (JAIR)*, 49:527–568, 2014.

[55] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. In *International Conference on Learning Representations*, 2018.

[56] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[57] Feng-hsiung Hsu. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, 2002.

[58] Aja Huang. AlphaGo Master online series of games, 2017. URL: `https://deepmind.com/research/alphago/match-archive/master`.

[59] Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Mohammadamin Barekatain, Simon Schmitt, and David Silver. Learning and planning in complex action spaces. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 4476–4486. PMLR, 18–24 Jul 2021.

[60] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[61] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z. Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[62] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. 11 2016.

[63] Wojciech Jaśkowski. Mastering 2048 with delayed temporal coherence learning, multi-state weight promotion, redundant encoding and carousel shaping. *IEEE Transactions on Computational Intelligence and AI in Games*, 04 2016.

[64] Philip N. Johnson-Laird. Mental models and human reasoning. *Proceedings of the National Academy of Sciences*, 107(43):18243–18250, 2010.

[65] Norman P. Jouppi, Cliff Young, Nishant Patil, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA ’17, pages 1–12. ACM, 2017.

[66] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H. Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model-based reinforcement learning for Atari. *CoRR*, abs/1903.00374, 2019.

[67] Tomoyuki Kaneko and Kunihito Hoki. Analysis of evaluation-function learning by comparison of sibling nodes. In *Advances in Computer Games - 13th International Conference, ACG 2011, Tilburg, The Netherlands, November 20-22, 2011, Revised Selected Papers*, pages 158–169, 2011.

[68] Steven Kapturowski, Georg Ostrovski, John Quan, Rémi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *ICLR*, 2019.

[69] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, page 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[70] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, page 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[71] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[72] Matthew Lai. Giraffe: Using deep reinforcement learning to play chess. Master's thesis, Imperial College London, 2015.

[73] Alexandre Laterre, Yunguan Fu, Mohamed Khalil Jabri, Alain-Sam Cohen, David Kas, Karl Hajjar, Torbjorn S. Dahl, Amine Kerkeni, and Karim Beguir. Ranked reward: Enabling self-play reinforcement learning for combinatorial optimization. *CoRR*, 2018.

[74] Yann LeCun, Y. Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[75] Yann LeCun and Yoshua Bengio. *Convolutional Networks for Images, Speech, and Time Series*, page 255–258. MIT Press, Cambridge, MA, USA, 1998.

[76] Shane Legg. Machine super intelligence. 2008.

[77] Siqi Liu, Kay Choong See, Kee Yuan Ngiam, Leo Anthony Celi, Xingzhi Sun, and Mengling Feng. Reinforcement learning for clinical decision support in critical care: Comprehensive review. *J Med Internet Res*, 22(7):e18477, Jul 2020.

[78] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *J. Artif. Int. Res.*, 61(1):523–562, jan 2018.

[79] Chris J. Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in Go using deep convolutional neural networks. In *International Conference on Learning Representations*, 2015.

[80] Amol Mandhane, Anton Zhernov, Maribeth Rauh, Chenjie Gu, Miaosen Wang, Flora Xue, Wendy Shang, Derek Pang, Rene Claus, Ching-Han Chiang, Cheng Chen, Jingning Han, Angie Chen, Daniel J. Mankowitz, Jackson Broshear, Julian Schrittwieser, Thomas Hubert, Oriol Vinyals, and Timothy Mann. Muzero with self-competition for rate control in vp9 video compression, 2022.

[81] John McCarthy. Ai as sport. *Science*, 276(5318):1518–1519, 1997.

[82] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[83] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

[84] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charlie Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[85] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning. *CoRR*, abs/1507.04296, 2015.

[86] Todd W. Neller. Pedagogical possibilities for the 2048 puzzle game. *J. Comput. Sci. Coll.*, 30(3):38–46, January 2015.

[87] Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[88] Kazuto Oka and Kiminori Matsuzaki. Systematic selection of n-tuple networks for 2048. In Aske Plaat, Walter Kosters, and Jaap van den Herik, editors, *Computers and Games*, pages 81–92, Cham, 2016. Springer International Publishing.

[89] OpenAI. OpenAI five. `https://blog.openai.com/openai-five/`, 2018.

[90] Sherjil Ozair, Yazhe Li, Ali Razavi, Ioannis Antonoglou, Aaron Van Den Oord, and Oriol Vinyals. Vector quantized models for planning. In Ma-

rina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 8302–8313. PMLR, 18–24 Jul 2021.

[91] Pascutto, Gian-Carlo and Linscott, Gary. Leela chess zero.

[92] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., USA, 1984.

[93] Barney Pell. A strategic metagame player for general chess-like games. *Computational Intelligence*, 12:177–198, 1996.

[94] J. Peters, S. Vijayakumar, and S. Schaal. Reinforcement learning for humanoid robotics. In *IEEE-RAS International Conference on Humanoid Robots (Humanoids2003)*, Karlsruhe, Germany, Sept.29-30, 2003.

[95] Tobias Pohlen, Bilal Piot, Todd Hester, Mohammad Gheshlaghi Azar, Dan Horgan, David Budden, Gabriel Barth-Maron, Hado van Hasselt, John Quan, Mel Večerík, et al. Observe and look further: Achieving consistent performance on Atari. *arXiv preprint arXiv:1805.11593*, 2018.

[96] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[97] Philip Rodgers and John Levine. An investigation into 2048 AI strategies. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–2, 2014.

[98] Tord Romstad, Marco Costalba, Joona Kiiski, et al. Stockfish: A strong open source chess engine. `https://stockfishchess.org/`. Retrieved November 29th, 2017.

[99] Christopher D. Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61:203–230, 2010.

[100] A. L. Samuel. Some studies in machine learning using the game of checkers II - recent progress. *IBM Journal of Research and Development*, 11(6):601–617, 1967.

[101] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *International Conference on Learning Representations*, Puerto Rico, 2016.

[102] Simon Schmitt, Matteo Hessel, and Karen Simonyan. Off-policy actor-critic with shared experience replay. *arXiv preprint arXiv:1909.11583*, 2019.

[103] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, and David Silver. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *Nature*, 588(7839):604–609, 2020.

[104] Julian Schrittwieser, Thomas Hubert, Amol Mandhane, Mohammadamin Barekatain, Ioannis Antonoglou, and David Silver. Online and Offline Reinforcement Learning by planning with a learned model. *Advances in Neural Information Processing Systems*, 34:27580–27591, 2021.

[105] Marwin Segler, Mike Preuss, and Mark Waller. Towards ”AlphaChem”: Chemical Synthesis Planning with Tree Search and Deep Neural Network Policies. 2017.

[106] Brian Sheppard. World-championship-caliber scrabble, 2002.

[107] David Silver. *Reinforcement Learning and Simulation-Based Search in Computer Go*. PhD thesis, CAN, 2009.

[108] 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. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[109] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[110] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[111] David Silver, Richard Sutton, and Martin Müller. Temporal-difference search in computer go. In *Machine Learning*, 2012.

[112] David Silver, Richard S. Sutton, and Martin Müller. Sample-based learning and search with permanent and transient memories. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, page 968–975, New York, NY, USA, 2008. Association for Computing Machinery.

[113] David Silver, Hado van Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David Reichert, Neil Rabinowitz, Andre Barreto, and Thomas Degris. The predictron: End-to-end learning and planning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3191–3199. PMLR, 06–11 Aug 2017.

[114] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[115] Richard Sutton. The bitter lesson, 2017. URL: `http://www.incompleteideas.net/IncIdeas/BitterLesson.html`.

[116] Richard S Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM SIGART Bulletin*, 2(4):160–163, 1991.

[117] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[118] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[119] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 2016.

[120] Marcin Szubert and Wojciech Jaśkowski. Temporal difference learning of n-tuple networks for the game 2048. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8, 2014.

[121] Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 2154–2162, Red Hook, NY, USA, 2016. Curran Associates Inc.

[122] Gerald Tesauro. TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[123] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Commun. ACM*, 38(3):58–68, March 1995.

[124] Gerald Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1-2):181–199, 2002.

[125] Gerald Tesauro and Gregory R. Galperin. On-line policy improvement using monte-carlo search. In *Advances in Neural Information Processing Systems 9*, pages 1068–1074, 1996.

[126] Tesla. Tesla AI Day. `https://www.youtube.com/watch?v=j0z4FweCy4M`, 2021.

[127] Sebastian Thrun. Learning to play the game of chess. In *Advances in neural information processing systems*, pages 1069–1076, 1995.

[128] Yuandong Tian and Yan Zhu. Better computer Go player with neural network and long-term prediction. 2015.

[129] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[130] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[131] J. Veness, D. Silver, A. Blair, and W. Uther. Bootstrapping from game tree search. In *Advances in Neural Information Processing Systems*, pages 1937–1945, 2009.

[132] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 2019.

[133] John von Neumann. *1. On the Theory of Games of Strategy*, pages 13–42. Princeton University Press, 2016.

[134] Yizao Wang and Sylvain Gelly. Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *2007 IEEE Symposium on Computational Intelligence and Games*, pages 175–182, 2007.

[135] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, may 1992.

[136] Kun-Hao Yeh, I-Chen Wu, Chu-Hsuan Hsueh, Chia-Chuan Chang, Chao-Chin Liang, and Han Chiang. Multistage temporal difference learning for 2048-like games. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(4):369–380, 2017.