Investigating AlphaZero's Value Targets Using Strongly Solved Chinese Checkers

by

Nicolas Ong

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

© Nicolas Ong, 2026

This work is licensed under CC Attribution 4.0 International

Abstract

AlphaZero learns to play games at a superhuman level through self-play, but many aspects of its training dynamics remain poorly understood. Existing analyses rely on indirect measures of value and policy quality such as win rates and Elo. In this thesis, we train AlphaZero on Chinese Checkers, a strongly solved game with ground truth values and policies available, to more directly investigate training dynamics. We (1) validate our AlphaZero implementation's learning ability, (2) discover causes of value head collapse and suggest mitigation strategies, and (3) investigate TD(λ) value targets, which yield a higher win rate and value head accuracy in the 25-6 and 49-4 settings than the default MC targets.

Preface

Nicolas Ong https://orcid.org/0009-0004-4620-3118

This thesis is an original work by Nicolas Ong. No part of this thesis has been previously published.

I acknowledge the use of generative artificial intelligence for assistance with writing and programming.

This research was supported by the Alberta Machine Intelligence Institute (Amii) and CIFAR. Computational resources, such as the Vulcan compute cluster, were provided by the Digital Research Alliance of Canada.

Acknowledgements

I thank my supervisor, Nathan Sturtevant, and Zaheen Ahmad for their guidance throughout my Master's degree.

I thank Bigyan Karki, whose work this thesis builds on.

I thank the members of my examining committee for their feedback that improved this thesis, and the members of the Moving AI Lab for advice and discussions that improved this work.

Table of Contents

Abstract ii

Preface iii

Acknowledgements iv

List of Tables ix

List of Figures x

List of Symbols xii

Chapter 1: Introduction 1

Chapter 2: Background & Related Work 5

2.1 Reinforcement Learning 5

2.1.1 Dynamic Programming 7

2.1.2 Monte Carlo & Temporal Difference Methods 8

2.1.3 n-step Returns & TD(λ) 8

2.2 Heuristic Search 9

2.2.1 Solving Games 10

2.2.2 Monte Carlo Tree Search 11

2.3 AlphaZero 13

2.4 Chinese Checkers 16

2.4.1 Rules 16

2.4.2 Solve Data 19

2.5 Related Work 20

2.5.1 AlphaZero 20

2.5.2 Analyzing & Improving AlphaZero 21

2.5.3 Analyzing & Improving AlphaZero with Ground Truth 24

Chapter 3: Validating AlphaZero on Chinese Checkers **27**

3.1 Establishing Neural Network Learning Capacity 27
3.1.1 Network Evaluation 28
3.1.1.1 Evaluation Targets 28
3.1.1.2 Evaluation Metrics 29
3.1.1.3 Standardizing State Sets 30
3.1.2 Network Training 31
3.1.2.1 Network Architecture 31
3.1.2.2 Training Procedure 31
3.1.3 Results 33
3.2 Establishing AlphaZero Learning Ability 36
3.2.1 AlphaZero Evaluation 36
3.2.1.1 Player Evaluation 36
3.2.1.2 Training Data Evaluation 37
3.2.2 AlphaZero Training 39
3.2.3 Results 40
3.2.4 Discussion and Related Work 42

Chapter 4: AlphaZero with Ground Truth **44**

4.1 Motivation & Background 44
4.2 Method 45
4.3 Results 45
4.4 Investigation 46
4.4.1 Value Head Prediction Distribution 46
4.4.2 Training Data Imbalance 47
4.4.3 Balancing the Training Data 48
4.5 Discussion & Related Work 50
4.5.1 Majority-Class Collapse Mitigations 50
4.5.1.1 Balancing the Dataset 50
4.5.1.2 Modifying the Loss Function 51
4.5.1.3 Thesis-Specific Mitigation Strategies 52

Chapter 5: AlphaZero with TD(λ) **54**
5.1 Motivation . . . 54
5.2 The TD(λ) Setting . . . 54
5.2.1 TD(λ) Targets . . . 54
5.2.2 TD(λ) Targets in AlphaZero . . . 56
5.2.3 TD(λ) Target Distribution . . . 56
5.3 TD(λ) Training Collapse Mitigation . . . 57
5.3.1 Dataset Balancing with Weighted Buckets . . . 57
5.3.1.1 Weighted Buckets Method . . . 57
5.3.1.2 Weighted Buckets in AlphaZero with TD(λ) . . . 58
5.3.1.3 Results of Weighted Buckets in AlphaZero with TD(λ) 58
5.3.2 Stabilizing the Value Head with Zero Initialization . . . 61
5.4 Method . . . 62
5.5 Results . . . 62
5.5.1 Value Head Accuracy . . . 62
5.5.2 Player Win Rate . . . 63
5.5.3 Value Target Accuracy . . . 72
5.6 Extensions to TD(λ) in AlphaZero . . . 72
5.6.1 TD(λ) Interpolation . . . 72
5.6.2 TD(λ) with Refreshes . . . 76
5.7 Discussion & Related Work . . . 79
5.7.1 Bias and Variance in TD(λ) . . . 79
5.7.1.1 Convergence Speed & Credit Propagation . . . 80
5.7.1.2 Final Model Quality & Variance over Game Length . 81
5.7.2 Future Work . . . 81
5.7.3 Related Work . . . 83

Chapter 6: Conclusion **85**

Bibliography **87**

Appendix A: Chinese Checkers Statistics **94**
A.1 Formulas . . . 94
A.2 Tables . . . 94

Appendix B: AlphaZero on Chinese Checkers **96**
B.1 Alternative Evaluation Methods 96
B.1.1 Targets 96
B.1.2 Evaluation Metrics 97
B.1.3 State Set Filters 98
B.1.4 Baseline Players 99
B.1.5 Player Evaluation Metrics 100
B.2 Hyperparameters 100
B.3 Neural Network Parameters 103
B.4 Additional Results 103
B.4.1 Effect of Model Size on Accuracy 103
B.4.2 Self-Play Statistics 103

Appendix C: AlphaZero with TD(λ) **105**
C.1 Policy Head Accuracy 105
C.2 Policy Target Accuracy 109
C.3 Hyperparameter Sweep 109

List of Tables

3.1 Training set sizes as a percentage of each state space 35

5.1 Converged value head accuracy across λ 63
5.2 Converged player win rate across λ 67

A.1 Solve data statistics for each board setting 95
A.2 Win/loss distribution for the player to move 95
A.3 Single-player statistics for each board setting 95

B.1 Supervised learning hyperparameters 101
B.2 AlphaZero hyperparameters 101
B.3 Player evaluation hyperparameters 102
B.4 Baseline player hyperparameters 102
B.5 Neural network parameter counts for each board size 103
B.6 Self-play statistics for each board size 104

C.1 Converged policy head accuracy across λ 105

List of Figures

2.1 The traditional Chinese Checkers board 16
2.2 Start positions for the four board sizes 17
2.3 Steps and hops in Chinese Checkers 18
2.4 Winning positions in Chinese Checkers 18
2.5 Illegal positions in Chinese Checkers 19

3.1 The AlphaZero network architecture 32
3.2 Supervised learning training curves 34
3.3 Final supervised learning accuracy versus training set size 35
3.4 Model accuracy on ground truth over training 41
3.5 Player win, loss, and draw rates against the baseline over training . . 42

4.1 Per-trial model accuracy with ground-truth targets 45
4.2 Value-head prediction distributions 46
4.3 Model accuracy with balanced ground-truth targets 49

5.1 Value-head accuracy distribution across TD(λ) and dataset balancing 59
5.2 Mean value prediction over training at $\lambda = 0$ across dataset balancing 60
5.3 Value head accuracy over training across TD(λ) (16-3) 64
5.4 Value head accuracy over training across TD(λ) (25-6) 65
5.5 Value head accuracy over training across TD(λ) (49-4) 66
5.6 Player win rate over training across TD(λ) (16-3) 68
5.7 Player win rate over training across TD(λ) (25-6) 69
5.8 Player win rate over training across TD(λ) (36-6) 70
5.9 Player win rate over training across TD(λ) (49-4) 71
5.10 Value target accuracy over training across TD(λ) (16-3) 73
5.11 Value target accuracy over training across TD(λ) (25-6) 74
5.12 Value target accuracy over training across TD(λ) (49-4) 75

5.13 Player win rate across k in TD(λ) with refreshes (36-6) 78
5.14 Per-trial player win rate under TD(λ) with refreshes (36-6) 79

B.1 Final supervised learning accuracy versus model size 104

C.1 Policy head accuracy over training across TD(λ) (16-3) 106
C.2 Policy head accuracy over training across TD(λ) (25-6) 107
C.3 Policy head accuracy over training across TD(λ) (49-4) 108
C.4 Policy target accuracy over training across TD(λ) (16-3) 110
C.5 Policy target accuracy over training across TD(λ) (25-6) 111
C.6 Policy target accuracy over training across TD(λ) (49-4) 112
C.7 Hyperparameter sweeps on TD(0) and TD(1) 113

List of Symbols

Background & Related Work

S Set of states in an MDP

A Set of actions in an MDP

$p(s'|s,a)$ Transition probability function

$r(s,a)$ Reward function

γ Discount factor

S_{term} Set of terminal states

$\pi(a|s)$ Policy: probability of taking action a in state s

G_t Return from time step t; the Monte Carlo value target

π_* Optimal policy; used as the ground-truth (uniform optimal) policy

$v_\pi(s)$ Value of state s under policy π

$q_\pi(s,a)$ Value of taking action a in state s under policy π

$v_*(s)$ Optimal value of state s; the game-theoretic (ground-truth) value from the solve data

$q_*(s,a)$ Optimal value of taking action a in state s

α Learning rate

δ_t TD error at time step t

$G_{t:t+n}$ n-step return from time step t

λ Interpolation parameter of TD(λ) targets

G_t^λ TD(λ) target: the λ-weighted sum of n-step returns

$Q(s, a)$ Value estimate of action a at state s in the MCTS tree

$N(s)$, $N(s, a)$ Visit counts of a node and of one of its children in the MCTS tree

c Exploration constant in the UCB1 and PUCB scores (reported as c_{puct})

$P(s, a)$ Prior policy used by the PUCT selection policy

f_θ Neural network, $f_\theta(s) = (\hat{v}_\theta(s), \hat{\pi}_\theta(s))$

θ, θ_i Neural network parameters, and parameters after training at iteration i

$\hat{v}_\theta$ Value head of the neural network

$\hat{\pi}_\theta$ Policy head of the neural network

$A(s)$ Set of legal actions in state s

π Policy target in a self-play tuple: the MCTS-derived distribution over $A(s)$

z Value target in a self-play tuple: the game outcome, $z \in \{-1, 0, 1\}$

(s, π, z) Self-play tuple

SP List of all self-play tuples generated during training

SP_i Self-play tuples generated during iteration i

$\mathcal{L}_v$, $\mathcal{L}_\pi$, $\mathcal{L}$ Value head, policy head, and total loss functions

P1, P2 The first and second player to move

Validating AlphaZero on Chinese Checkers

$A^*(s)$ Set of optimal actions in state s

s_a State reached by taking action a in state s

$\text{Acc}(\hat{v} \mid v_*, S)$ Value accuracy of $\hat{v}$ on state set S

$\tilde{v}_\theta$ Discretized (± 1) output of the value head

$\mathrm{Acc}(\hat{\pi} \mid \pi_*, S)$ Policy accuracy of $\hat{\pi}$ on state set S

$\Phi_v(S)$, S^v Value head filter: removes draw states and balances winning and losing states

$\Phi_\pi(S)$, S^π Policy head filter: removes trivial states

S_{train}, S_{eval} Training and evaluation state sets for supervised learning

$\mathrm{DistEval}_i(s)$ Distance-based heuristic evaluation of s for player i, used by the baseline player

SP^v Self-play tuples with $z \neq 0$

SP^π Self-play tuples excluding trivial states

$\rho(s)$ Proportion of optimal actions, $|A^*(s)|/|A(s)|$

$\mathrm{Acc}(\text{SP} \mid v_*)$ Value accuracy of the self-play tuples in SP

$\mathrm{Acc}(\text{SP} \mid \pi_*)$ Policy accuracy of the self-play tuples in SP

S_{random} Evaluation set of states sampled uniformly from the state space

S_{seen} Evaluation set of states seen during self-play

$S_{\text{neighbor1}}$, $S_{\text{neighbor2}}$ Evaluation sets of states one and two moves away from states in S_{seen}

AlphaZero with Ground Truth

$\text{SP}_{*,i}$ Self-play tuples at iteration i with targets replaced by ground truth, $\{(s, \pi_*(s), v_*(s))\}$

p_{maj} Proportion of the majority class among ± 1 value targets

AlphaZero with TD(λ)

SP_i^λ Self-play tuples at iteration i with TD(λ) value targets

n Number of buckets used by the weighted buckets method

k Number of epochs, and of target refreshes, per iteration

Appendices

N Number of board locations

p Number of pieces per player

$d_*(s)$ Distance to win from s under optimal play

$v_{*\gamma}(s)$ Discounted ground-truth value, $\gamma^{d_*(s)} v_*(s)$

$\pi_{*\gamma}$ Ground-truth policy derived from $v_{*\gamma}$, which prefers faster wins

$D_{\mathrm{KL}}(\pi_* \parallel \hat{\pi})$ KL divergence between the ground-truth and predicted policies

$H(\pi_*, \hat{\pi})$ Cross entropy between the ground-truth and predicted policies

$A^*_\pi(s)$ Set of actions most preferred by policy π in state s

DBEval Database-backed heuristic evaluation, an exact form of DistEval

Chapter 1

Introduction

Artificial Intelligence (AI) endeavours to build computer programs able to emulate functions of human intelligence, such as recognizing images, translating speech, and making decisions [1]. Core to this is the ability to accomplish goals when acting in a given environment [2, 3]. Games such as Chess, Go, and Shogi have been used as environments to test AI ability due to their well-defined rules and the complexity of decision-making they require. AlphaZero, an AI system developed by DeepMind, has demonstrated superhuman performance in these games [4]. However, understanding the training dynamics of AlphaZero and how it achieves such performance remains a challenge.

Current approaches to understanding AlphaZero's training dynamics are limited by the lack of ground truth data. Analyses rely on metrics such as win rates and Elo ratings, which are expensive to collect and only measures performance indirectly. Despite these limitations, such analyses have provided insights which have brought improvements to the training process, including engineered input features, auxiliary targets, novel neural network architectures, and other new training methods [5–9]. These analyses have also demonstrated the existence of non-transitive strategies, such as local improvements that do not generalize well to other strategies and adversarial strategies that exploit weaknesses in the training process [5, 10].

In this thesis, we use ground truth to analyze, better understand, and improve AlphaZero's training dynamics. Ground truth, when available, enables inexpensive and direct measurement of player performance. However, ground truth is difficult to collect in complex environments. We are able to use ground truth because we train AlphaZero in Chinese Checkers, a game that has been strongly solved for board sizes up to 6×6 with 6 pieces (36-6) [11]. Previous work analyzing AlphaZero with ground truth exists. Ground truth has been used to analyze how various hyperparameters and training strategies affect the accuracy of the model trained by AlphaZero [12], to analyze the behaviour of MCTS in AlphaZero [13, 14], and to analyze how adversarial strategies exploit AlphaZero players [15]. This thesis expands on this existing work.

Our main contributions are as follows.

- **An AlphaZero implementation for strongly solved Chinese Checkers, together with a ground truth evaluation methodology.** We define metrics that measure the accuracy of the value head, the policy head, and the self-play training data using ground truth. We use them to confirm prior observations that the value head attains a higher accuracy than the training data it learns from [15].

- **A diagnosis of why ground truth value targets prevent AlphaZero from learning.** We replicate the negative result of Karki *et al.* [16] and identify its cause as majority-class collapse in the value head. We show that AlphaZero's default Monte Carlo targets are balanced by construction, so any replacement target risks introducing an imbalance that the standard training process cannot absorb.

- **Mitigation strategies that make alternative value targets viable in AlphaZero.** We adapt two standard majority-class collapse mitigations to

AlphaZero, majority-class subsampling and class weighting, and extend the latter into weighted buckets to handle continuous targets. We also adapt zero-initialization of the value head, drawn from work on training stability rather than class imbalance. These strategies enable AlphaZero to learn from alternative value targets, including ground truth and TD(λ) targets. We recommend addressing instability before data imbalance because a stable network is robust to the imbalance that alternative targets produce in practice.

- **An evaluation of TD(λ) value targets in AlphaZero.** Lower values of λ yield more accurate training data, and TD(λ) targets yield stronger final models by both value head accuracy and win rate, at the cost of slower convergence, compared to the default Monte Carlo targets. We propose two extensions: TD(λ) interpolation, which recovers the convergence speed of Monte Carlo targets while retaining the final model quality of TD(λ), and TD(λ) with refreshes, which also converges quickly but less stably and provides empirical support for slow credit assignment as the explanation for TD(λ)'s slower convergence.

The remainder of this thesis is organized as follows.

In Chapter 2 (Background & Related Work), we provide background on TD(λ), AlphaZero, and the game of Chinese Checkers. We also provide an overview of previous work analyzing and improving AlphaZero, both with and without ground truth.

In Chapter 3 (Validating AlphaZero on Chinese Checkers), we (1) define the evaluation metrics we use to analyze AlphaZero throughout the thesis, (2) validate our neural network's ability to learn by training it on ground truth with supervised learning, and (3) validate our AlphaZero implementation's learning ability by measuring an increase in accuracy and win rate over training.

In Chapter 4 (AlphaZero with Ground Truth), we replicate an experiment from previous work [16] that replaced the default targets in AlphaZero with ground truth

and found that this prevented the AlphaZero player from learning. We then find the cause of this failure to be majority-class collapse in the value head caused by imbalanced training data and an unstable training process. We develop mitigation strategies to prevent this collapse, and find that these strategies allow the AlphaZero player to learn with ground truth targets.

Finally, in Chapter 5 (AlphaZero with TD(λ)), we investigate the use of TD(λ) value targets in AlphaZero. We first create two majority-class collapse mitigation strategies for the TD(λ) setting. We then compare the training process of AlphaZero with TD(λ) targets over multiple values of λ. We then explore two extensions to the TD(λ) setting, TD(λ) interpolation and TD(λ) with refreshes.

Chapter 2

Background & Related Work

In this chapter, we provide the background concepts relevant to this thesis' goal, which is to analyze AlphaZero's training dynamics using ground truth to better understand and improve it. Reinforcement Learning provides the theoretical framework for algorithms which attempt to learn to play games, including AlphaZero. Neural Networks are used to approximate the value and policy functions in AlphaZero. Heuristic Search is used to both generate solve data that can be used as ground truth and select actions in AlphaZero. We provide an overview of the rules of Chinese Checkers, which is the game we focus on in this thesis, and the solve data available for it. Finally, we provide an overview of related work in analyzing AlphaZero.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a type of machine learning where an agent learns to maximize rewards while interacting with an environment. RL is important for this thesis because it establishes the theoretical framework for most algorithms that learn to play games, including AlphaZero. In this section, we provide an overview of foundational RL concepts and value-based methods from Sutton and Barto [3], which are relevant throughout this thesis.

Environments in RL are often modeled as Markov Decision Processes (MDPs).

MDPs can be defined by the tuple (S, A, P, R, γ), where S is a set of states, A is a set of actions, $P = p(s'|s, a)$ is the transition probability function which defines the probability of transitioning to state s' given the current state s and action a, and $R = r(s, a)$ is the reward function which assigns a reward to each state-action pair. $\gamma \in [0, 1]$ is the discount factor, which determines the importance of future rewards.

Agents interact with the environment by starting at a starting state s_0, then taking an action a_0 according to its policy π. The environment responds with a new state s_1 and a reward r_1. In episodic RL, this process repeats until a terminal state is reached, generating a sequence $(s_0, a_0, r_1, s_1, a_1, r_2, \ldots, a_{T-1}, r_T, s_T)$. A terminal state is a state $s \in S_{\text{term}} \subseteq S$ such that upon entering any $s \in S_{\text{term}}$, the episode ends. The policy $\pi : S \to \Delta(A)$ maps each state to a probability distribution over actions, where $\pi(\cdot|s)$ denotes the distribution at state s, and $\pi(a|s)$ gives the probability of selecting action a in state s, such that $\sum_{a \in A} \pi(a|s) = 1 \quad \forall s \in S$.

In each episode, the agent receives a sequence of rewards $r_1, r_2, \ldots, r_T$. From this, we can define the discounted return at time t as $G_t = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1}$, which is the sum of discounted rewards from time step t to the end of the episode. The goal of the agent is to learn an optimal policy π_* that maximizes the expected return over time.

The value function $v_\pi(s)$ denotes the expected return, or value, when starting at state s and following policy π. The action-value function $q_\pi(s, a)$ denotes the expected return when taking action a in state s, and thereafter following policy π. The optimal value function $v_*(s)$ and optimal action-value function $q_*(s, a)$ correspond to an optimal policy π_*, which achieves the highest expected return possible from state s and state-action pair (s, a), respectively. An optimal policy is guaranteed to exist. However there may be multiple optimal policies.

2.1.1 Dynamic Programming

The Bellman equations express a recursive relationship between the value of a state (or state-action) and its successor states (or state-actions). The Bellman equations can be written as:

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2.1}$$

$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a) \sum_{a'} \pi(a'|s')[r + \gamma q_\pi(s', a')] \tag{2.2}$$

Dynamic Programming (DP) methods are a set of algorithms that can compute an optimal policy π_* and its corresponding value function v_* or q_* by leveraging the Bellman equations. DP methods require a model of the environment's dynamics (i.e., the transition probabilities and reward function) to compute.

Policy Iteration is one such DP method. It works by iteratively evaluating and improving a policy through two main steps: policy evaluation and policy improvement. In the policy evaluation step, the value function v_π for the current policy π is computed. To do this, the Bellman equations are repeatedly applied to an arbitrary initial value function (or the previous iteration's value function) until convergence. In the policy improvement step, an improved policy π' is created using the computed value function. For example, if q_π was computed, $\pi'(s) := \text{argmax}_a\, q_\pi(s, a)$ guarantees a stronger policy than π. Starting with an arbitrary policy π_0, the policy evaluation step computes v_{π_0}. Then the policy improvement step generates a new policy π_1. This process repeats, generating a sequence of policies and value functions π_0, v_{π_0}, π_1, v_{π_1}, π_2, v_{π_2}, ..., π_k, v_{π_k}, until the policy stabilizes and converges to the optimal policy π_*.

Generalized Policy Iteration (GPI) is a more general form of this, covering any method where policy evaluation and policy improvement steps interact to learn a better policy. For example, methods where the policy evaluation step is not run to

convergence or where policy evaluation uses a different method of value estimation are also covered by GPI. Many RL methods for learning π_* fall under the GPI framework.

2.1.2 Monte Carlo & Temporal Difference Methods

DP methods require a model of the environment's dynamics, which is not always available. Monte Carlo (MC) and Temporal Difference (TD) methods are model-free methods which learn by interacting with the environment.

MC methods estimate $v_\pi(s)$ by averaging the returns (G_t) observed after visiting s over multiple episodes, or estimating $q_\pi(s, a)$ by averaging the returns observed after taking action a in state s. MC methods are unbiased but have high variance. MC methods need complete episodes to compute returns.

TD methods allow for updating value estimates without waiting for the end of an episode. One such TD method is TD(0), which updates the value of a state s_t based on the observed reward r_{t+1} and the estimated value of the next state s_{t+1}. This is done using the update rule:

$$v_\pi(s_t) \leftarrow v_\pi(s_t) + \alpha[r_{t+1} + \gamma v_\pi(s_{t+1}) - v_\pi(s_t)] \tag{2.3}$$

where α is the learning rate, $r_{t+1} + \gamma v_\pi(s_{t+1}) - v_\pi(s_t)$ is the TD error δ_t, and $r_{t+1} + \gamma v_\pi(s_{t+1})$ is the TD target: the one-step bootstrapped estimate of $v_\pi(s_t)$. This replaces the full return G_t that MC would use. TD methods are biased but have lower variance than MC methods.

2.1.3 n-step Returns & TD(λ)

n-step returns create a spectrum of targets between the MC target G_t and the TD(0) target $r_{t+1} + \gamma v_\pi(s_{t+1})$. The n-step return $G_{t:t+n}$ is defined as:

$$G_{t:t+n} = \left(\sum_{k=0}^{n-1} \gamma^k r_{t+k+1} \right) + \gamma^n v_\pi(s_{t+n}) \tag{2.4}$$

If the n-step return extends to termination, then it is equal to the full return ($G_{t:t+n} = G_t$ if $t + n = T$, where T is the time of termination).

TD(λ) combines n-step returns into a single target by taking a weighted sum of all n-step returns. The TD(λ) target is defined as:

$$G_t^\lambda = (1 - \lambda) \left(\sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} \right) + \lambda^{T-t-1} G_t \tag{2.5}$$

where $\lambda \in [0, 1]$ is a hyperparameter that controls the weighting of the different n-step targets. When $\lambda = 0$, the TD(λ) target G_t^λ reduces to the TD(0) target, $r_{t+1} + \gamma v_\pi(s_{t+1})$. When $\lambda = 1$, G_t^λ reduces to the MC target G_t. Values of λ between 0 and 1 interpolate between these two targets. Eligibility traces (and Dutch traces) enable TD(λ) to remain an online learning method where, rather than waiting for the end of an episode to compute G_t^λ, the estimated value of s_t can be updated with each new reward and state observed.

In this thesis, we use offline methods and ignore discounting.

2.2 Heuristic Search

Heuristic search uses heuristics to guide search in a combinatorially large search space. In the context of games, heuristic search algorithms are used to explore the game tree, where nodes can represent game states and edges represent actions. Common goals include solving games and finding good moves given a limited search budget. Both these applications of search are relevant to this thesis, as search can generate ground truth data for evaluating AlphaZero, and can help select actions during self-play in AlphaZero.

2.2.1 Solving Games

The standard taxonomy for solving games [17] is as follows:

- **Ultra-weakly solved**: For the initial position(s), the game-theoretic value has been determined.
- **Weakly solved**: For the initial position(s), the game-theoretic value and a strategy to obtain this value has been determined.
- **Strongly solved**: For all legal positions, the game-theoretic value and a strategy to obtain this value has been determined.

This thesis focuses on two-player, deterministic, zero-sum games with perfect information. These are games where two players alternate moves, and the full state of the game is known to both players at all times. Zero-sum means that the players' payoffs sum to zero at every terminal state, so a payoff of u for one player corresponds to a payoff of $-u$ for the other. Typically, $u \in \{-1, 0, 1\}$, where 1 is a win, -1 is a loss, and 0 is a draw. A loss (-1) for one player is a win (1) for the other.

We adopt the convention that the game-theoretic value of a state is given from the perspective of the player to move. This is the negamax convention. There is also the minimax convention, where the game-theoretic value is given from the perspective of a single player. Under the negamax convention, the game-theoretic value of a state s is the value that the player to move obtains when both players play optimally from s onwards.

Solving games can provide additional metrics, such as distance-to-win (DTW) information: the number of moves to achieve the game-theoretic value under optimal play, where the winning player minimizes this distance and the losing player maximizes it. This convention originates in chess endgame tablebases, where the metric is known as depth-to-mate (DTM) [18, 19].

Some examples of weakly solved games include Checkers [20], Gomoku [21], Othello [22], and 5×5 Go [23]. Strongly solved games give us a ground truth for all states to evaluate players against. This is useful for this thesis, where we want to evaluate AlphaZero's performance, investigate its learning dynamics, and possibly improve it. DTW information can also be useful for establishing a ground truth for evaluating players.

Some examples of strongly solved games include Tic-Tac-Toe, Awari [24], Pentago [25], Connect Four [26, 27], and Chinese Checkers [11]. Many games of interest, especially games that are popular and have both complexity and strategic depth, remain unsolved or are only partially solved. For example, Chess and Go are not strongly solved. The large number of states in complex games makes it difficult to compute the game-theoretic value for all positions [28].

Many games have partial solutions, such as endgame tablebases for Chess [18, 29], or exact solvers that can solve specific game states with reasonable resources, such as in Hex [30] and Go [31]. Other games can be parameterized by board size, with solve data only provided for lower board sizes [11, 30].

2.2.2 Monte Carlo Tree Search

Classic search algorithms for solving games, such as minimax search with alpha-beta pruning, have also been used with bounded search budgets and heuristic evaluation functions to select moves [32, 33]. However, exhaustive tree search algorithms become computationally infeasible in games with large branching factors or deep game trees [34, 35]. One approach to handling this limitation is Monte Carlo Tree Search (MCTS) [36], where tree search is combined with Monte Carlo methods to approximate solutions where exhaustive search is intractable.

MCTS can be broken down into four steps, which are repeated until a stopping

condition (iteration limit or time limit) is met.

1. **Selection**: Starting from the root node, select a child node based on a selection policy until a leaf node is reached.

2. **Expansion**: Expand the selected leaf node by adding its child nodes – unless it is a terminal state.

3. **Simulation**: Perform a random rollout from the selected node to a terminal state to obtain a value estimate.

4. **Backpropagation**: Propagate the value estimate back up the tree, updating the statistics of the nodes along the path.

The selection policy needs to balance exploration and exploitation during the selection step. Exploitation selects the best child node based on its current value estimate (the player to move maximizes), while exploration promotes visiting less-visited child nodes. This may lead to discovering better moves than the ones currently being exploited. Without exploration, the search may get stuck on actions whose early value estimates were optimistic or miss actions with pessimistic early value estimates. A common selection policy to accomplish this is UCT (UCB (Upper Confidence Bounds) applied to trees) [37]. The UCT selection policy selects the child node corresponding to action a from state s that maximizes the UCB1 score:

$$UCB1(s,a) = Q(s,a) + c \cdot \sqrt{\frac{\ln N(s)}{N(s,a)}} \tag{2.6}$$

where $Q(s,a)$ is the value estimate of the child node corresponding to action a from state s, $N(s)$ is the visit count of the parent node, $N(s,a)$ is the visit count of the child node, and c is a hyperparameter that controls the balance between exploration and exploitation.

The selection policy can also incorporate a prior policy, which is a probability distribution over actions that indicates which actions are more promising. This can be done by modifying the UCB1 score to include a prior policy $P(s, a)$. This creates the PUCT (Predictor + UCT) selection policy [38], which selects the child node corresponding to action a from state s that maximizes the PUCB score:

$$PUCB(s, a) = Q(s, a) + c \cdot P(s, a) \cdot \frac{\sqrt{N(s)}}{1 + N(s, a)} \tag{2.7}$$

The purpose of the simulation step is to acquire a value estimate for the selected node. The random rollout is one high variance and low bias option to obtain this estimate. However, there are other options. A heuristic function can be used, like a neural network trained to predict the value of a state. If such a heuristic is available, rollouts can be terminated once some step budget is reached, and the heuristic then replaces (or supplements) the simulation outcome as the value estimate. Further, the rollout itself can be guided by a policy.

Once an estimate is obtained, it is propagated back through the tree using backpropagation. Backpropagation updates the visit count ($N(s)$) and value estimate ($Q(s)$) for each node along the path from the selected node back to the root. $N(s)$ is the number of times the node has been visited during the selection step. $Q(s)$ is the average of the rollout values backpropagated through the node.

2.3 AlphaZero

AlphaZero [4] learns to play sequential two-player perfect-information zero-sum games through repeated iterations of self-play and training. In the self-play phase of each iteration, the current model plays games against itself to generate training data. In the training phase of each iteration, the model is trained on this data to improve its performance.

AlphaZero attempts to predict the value and optimal policy for each state using a neural network $f_\theta(s) = (\hat{v}_\theta(s), \hat{\pi}_\theta(s))$, where $\hat{v}_\theta(s)$ is the model's value estimate of state s and $\hat{\pi}_\theta(s)$ is the model's policy over legal actions in state s. The goal of training is for $\hat{v}_\theta(s)$ and $\hat{\pi}_\theta(s)$ to converge to the target value $v(s)$ and policy $\pi(s)$, respectively. We denote the model parameters at iteration i as θ_i. The model is initialized with parameters θ_0. $\hat{v}_{\theta_i}$ and $\hat{\pi}_{\theta_i}$ denote the value head and policy head after training at iteration i, respectively.

During self-play, the current model plays against itself to generate training data. At each state s, the model uses MCTS to select actions, where the policy head $\hat{\pi}_\theta(s)$ is used as a prior in the PUCT selection policy and the value head $\hat{v}_\theta(s)$ is used when evaluating states. A policy vector $\pi \in \mathbb{R}^{|A(s)|}$ over the legal actions for state s, $A(s)$, is generated by applying softmax to the visit counts of each child node of the root in the MCTS search tree. Dirichlet noise is added to the policy vector to encourage exploration, then an action is sampled and played in the game. This process repeats until a terminal state is reached, at which point the outcome of the game $z \in \{-1, 0, 1\}$ is observed.

At the end of each game, the self-play data is stored as a list of tuples (s, π, z) for each state s visited during the game. π is the policy vector generated by the MCTS search, and z is the outcome of the game in the perspective of the player to move in state s. We denote the list of all self-play tuples generated during training as SP, and SP_i as the list of self-play tuples generated during iteration i.

The self-play tuples are stored in a replay buffer, which is a data structure that holds a fixed number of the most recent self-play tuples. When the buffer is full, the oldest tuples are discarded to make room for new ones. The replay buffer allows the model to learn from a diverse set of experiences, rather than just the most recent games.

During training, the model is trained on mini-batches of self-play tuples sampled from the replay buffer. z is the value head target, and π is the policy head target. The loss function for the value head is Mean Squared Error (MSE) between the predicted value $\hat{v}_\theta(s)$ and the actual outcome z:

$$\mathcal{L}_v(\hat{v}_\theta(s), z) = (\hat{v}_\theta(s) - z)^2 \tag{2.8}$$

The loss function for the policy head is the Cross-Entropy (CE) loss between the predicted policy $\hat{\pi}_\theta(s)$ and the MCTS-generated policy π:

$$\mathcal{L}_\pi(\hat{\pi}_\theta(s), \pi) = -\sum_{a \in A(s)} \pi(a) \log \hat{\pi}_\theta(a|s) \tag{2.9}$$

The overall loss function for training the model is a sum of the value and policy losses:

$$\mathcal{L}(\hat{v}_\theta(s), z, \hat{\pi}_\theta(s), \pi) = \mathcal{L}_v(\hat{v}_\theta(s), z) + \mathcal{L}_\pi(\hat{\pi}_\theta(s), \pi) \tag{2.10}$$

In RL terms, AlphaZero is a value-based method that fits in the GPI framework. The policy improvement step is performed by MCTS, which uses the current value estimate $\hat{v}_\theta$ (and current policy $\hat{\pi}_\theta$) to generate a stronger policy π. The policy evaluation step is performed by MC methods, playing full games to estimate the value of each state. The current policy and value estimates are stored and approximated in the neural network f_θ. AlphaZero starts with arbitrarily initialized $\hat{v}_{\theta_0}$ and $\hat{\pi}_{\theta_0}$. The policy improvement step (MCTS) results in new policy head targets which results in $\hat{\pi}_{\theta_1}$. The policy evaluation step (self-play) results in new value head targets which results in new value head parameters $\hat{v}_{\theta_1}$. The policy improvement and policy evaluation steps are repeated, generating a sequence of policies and value functions f_{θ_i}, $f_{\theta_{i+1}}$, until convergence.

2.4 Chinese Checkers

Chinese Checkers is a zero-sum game with perfect information. We focus on Chinese Checkers in this thesis because it is a complex game similar to Chess and Go, but has been strongly solved for smaller board sizes. This allows us to evaluate AlphaZero against ground truth. In this section, we describe the rules of Chinese Checkers and the solve data available for evaluation.

2.4.1 Rules

Chinese Checkers is traditionally played on a hexagram-shaped board with 121 holes. There are up to 6 players, each with 10 pieces which start in one of the six triangular starting areas. The objective is to move all of one's pieces into the opposite triangle before the opponent does. The traditional board and starting positions are shown in Figure 2.1.

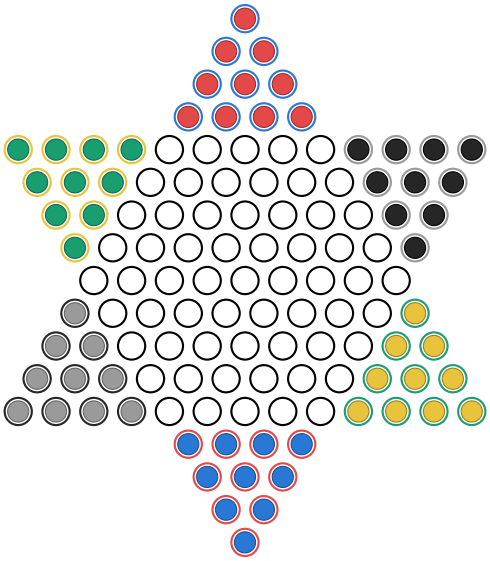
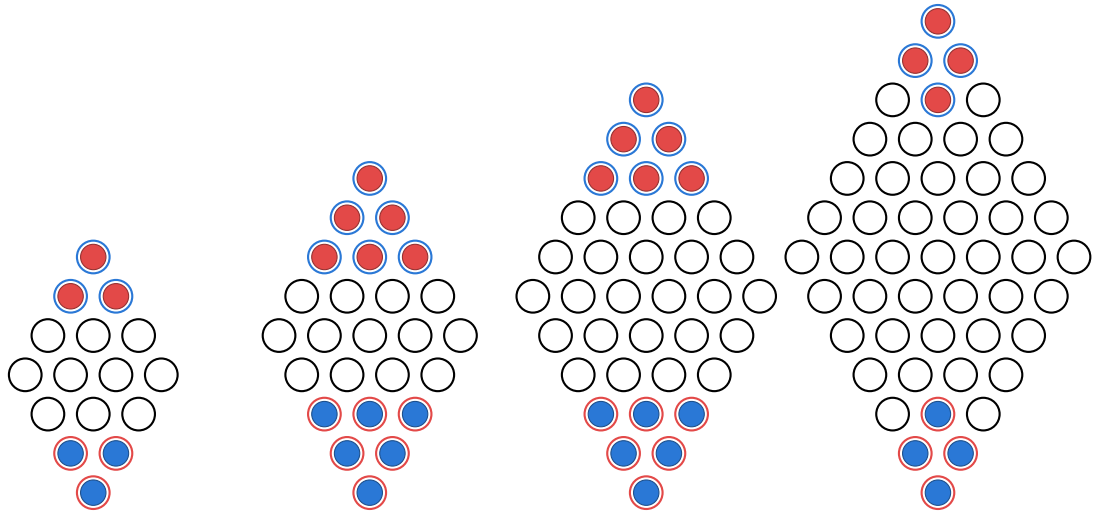


Figure 2.1: The traditional Chinese Checkers board. Fill colors indicate the starting positions of each player's pieces. Outline colors indicate each player's goal area.

In this thesis, we follow the rules of the two-player variant of Chinese Checkers

as described by Sturtevant [11]. In this variant, the game is played on an $n \times n$ rhombus-shaped board. Both players start with their pieces in opposite corners, and take turns moving one piece at a time. The goal of each player is to move their pieces into the opponent's starting area. The first player to move is denoted P1, and the second player is denoted P2.

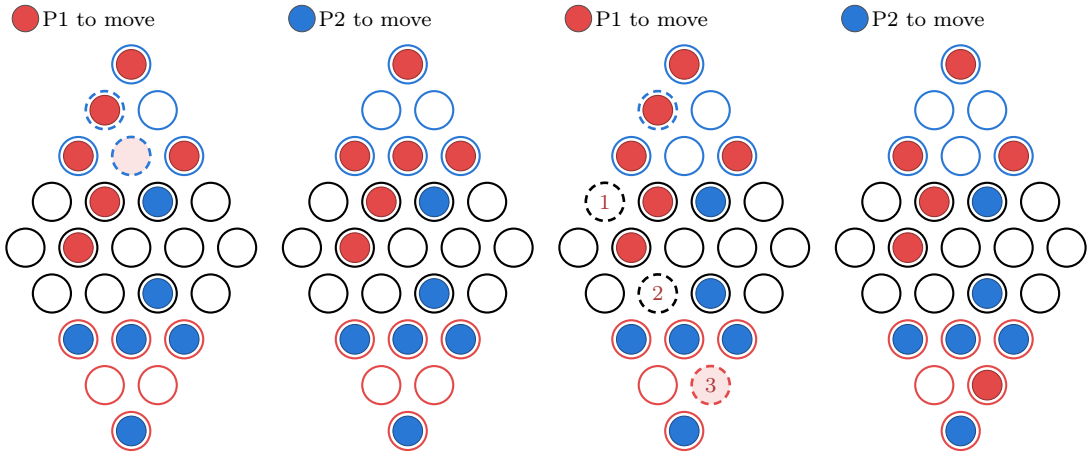


(a) 16-3 (b) 25-6 (c) 36-6 (d) 49-4

Figure 2.2: Start positions for the four board settings used in this thesis. P1 (red) starts in the top corner and P2 (blue) in the bottom corner. Position outline color indicates which player the goal area belongs to. Fill color indicates which player's pieces are in the position.

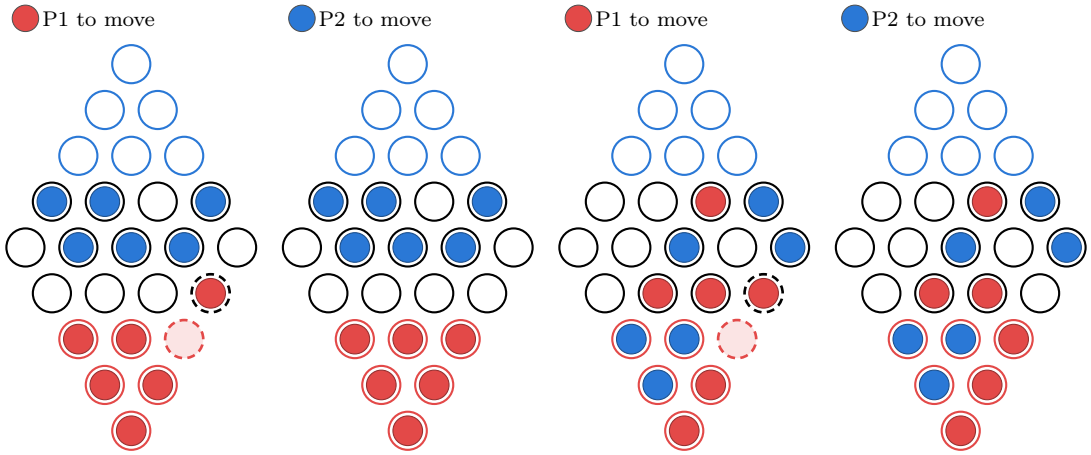
The board size and number of pieces each player has can be varied. This thesis focuses on the following board sizes and number of pieces: 4×4 with 3 pieces (16-3), 5×5 with 6 pieces (25-6), 6×6 with 6 pieces (36-6), and 7×7 with 4 pieces (49-4). The starting positions for each setting are shown in Figure 2.2.

There are two types of moves available to players: steps and hops. Steps involve moving a piece to an adjacent empty location, while hops allow a piece to jump over an adjacent occupied location. Multiple hops can be chained together in a single turn. Figure 2.3 gives an example of a step move and a hop move (with chaining).



(a) Step; Before (b) Step; After (c) Hops; Before (d) Hops; After

Figure 2.3: Demonstration of the two move types in Chinese Checkers. (a) and (b) show a step move. (c) and (d) show a hop move, where 3 hops are chained together in one move. To indicate the move being taken, positions it touches are given a dashed outline and its final position is shaded. Additionally, chained hops have their path indicated with numbering.



(a) (b) (c) (d)

Figure 2.4: Demonstration of the two ways P1 can fill its goal area to win. (a) and (b) show P1 fill its goal area with its own pieces. (c) and (d) show P1 fill its goal area with a mix of its own pieces and P2's pieces.

A player wins the game if their goal area (equivalent to their opponent's starting area) is filled with pieces, at least one of which is the player's own piece. A game ends in a draw if a board state is repeated during the game. Figure 2.4 shows two examples of winning moves for P1.

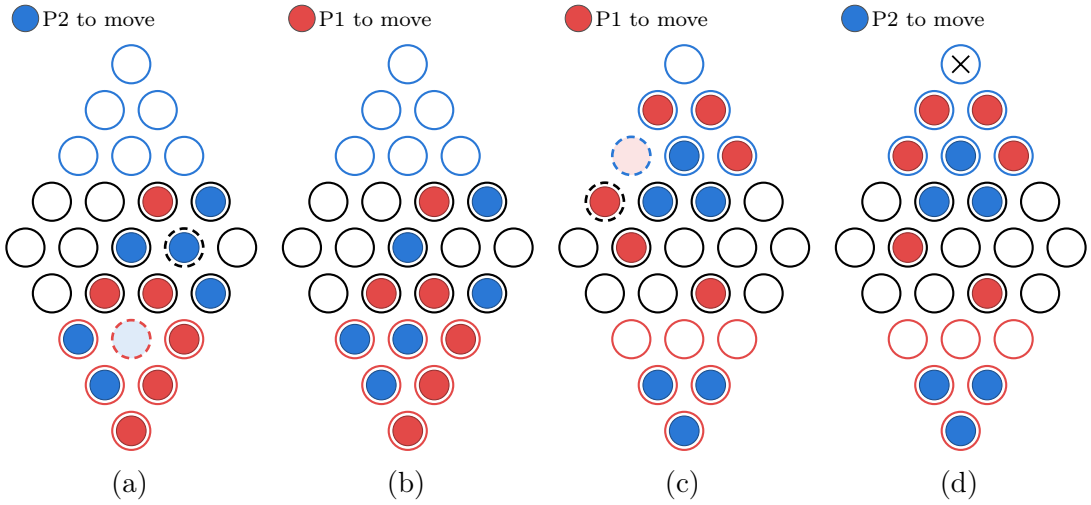


Figure 2.5: Demonstration of two moves that result in an illegal state. (a) and (b) show a move that is suicidal. The state it leads to is declared illegal rather than won. (c) and (d) show a move that creates a wall preventing P2 from reaching a position in its goal area. This state is illegal whichever player is to move.

Chinese Checkers also has illegal states. A state is illegal if (1) the winning conditions are met for player n and it is player n's turn to move (i.e. it is illegal for a player to take a suicidal action), or (2) there are unoccupied locations in player n's goal area that are unreachable by player n due to the presence of the opponent's pieces (i.e. it is illegal for a player to block their opponent's pieces from reaching any location in their goal area). Figure 2.5 shows these two types of illegal states.

2.4.2 Solve Data

Chinese Checkers has been strongly solved for board sizes of up to 6×6 with 6 pieces [11], a board size with about 2 trillion states. For comparison, the original board size of 9×9 with 10 pieces has about 1.7×10^{24} states. However, we only had

access to the solve data for 16-3, 25-6, and 49-4 because the 36-6 solve data was too large for our scripts to handle. The 36-6 solve data is about 580GB (or 145GB by exploiting symmetries).

The solve data for Chinese Checkers labels each state as either a win, a loss, a draw, or illegal. The win/loss labels are from the perspective of the maximizing player (P1), but can be converted to the perspective of the player to move. The solve data does not include distance-to-win (DTW) information.

2.5 Related Work

2.5.1 AlphaZero

AlphaZero [4] was introduced in 2017 following an arc of previous work. AlphaGo [34] established the MCTS with self-play architecture of AlphaZero. It initialized its policy network by performing supervised learning on 30 million positions from the KGS Go Server, then performed the self-play training procedure on top of that foundation. AlphaGo defeated Lee Sedol, the winner of 18 international titles, in March 2016 [39]. AlphaGo Zero [39] learns tabula rasa, meaning it is able to learn through just the self-play training procedure, without human data to start off with. Finally, AlphaZero [4] expanded the scope of the algorithm to other games. It has been succeeded by MuZero [40], which is able to learn a model of the environment's dynamics, enabling it to perform search without being given the dynamics of the environment.

Despite being introduced in 2017 [41] (9 years ago as of 2026), AlphaZero remains the basis for many strong neural-network game-playing programs. In Go, an open-source program based on AlphaZero, KataGo [6], is one of the strongest Go-playing programs available. As of August 2026, KataGo-based players occupy leading positions on the Computer Go Server (CGOS) 9×9, 13×13, and 19×19 rating lists [42–44]. In Chess, Leela Chess Zero (Lc0) [45], an open-source program based on AlphaZero, won

the Top Chess Engine Championship (TCEC) superfinal in Seasons 15 (2019) [46] and 17 (2020) [47], and was runner-up to Stockfish in Seasons 14 (2019), 18-21 (2020-2021), and 23-28 (2022-2025) [48]. Stockfish, an alpha–beta searcher that uses an NNUE [49] evaluation, has held the title continuously since Season 18 [48]. Both KataGo and Lc0 are based on the AlphaZero architecture, but have added various improvements to the original algorithm.

AlphaZero has since been trained on Go [4, 50], Chess [4], Shogi [4], Othello [50], Connect Four [12], Chinese Checkers [15], and various other games.

2.5.2 Analyzing & Improving AlphaZero

There is existing research that focuses on analyzing and improving AlphaZero. We discuss some of the relevant observations and improvements here.

Neural Network Architecture Some improvements to the neural network architecture have been proposed. For example, ELF OpenGo [5] observes that AlphaZero struggles to learn "ladder moves" – scenarios which span the whole board – in Go. They suspect this is because CNNs lack the right inductive biases. One remedy to this may be to add "Global Pooling" to CNNs, done in KataGo [6], which adds global context to the convolutional net.

KataGo [6] also uses progressive net sizing, where they start training with a smaller model to speed up training and inference. They increase the model size once the smaller model saturates, and repeat until they reach the desired model size. This saves time during the early stages of training.

Augmented Inputs & Targets KataGo [6] adds auxiliary targets which adds some regularization functionality. They prevent the NN from overfitting and encourage better representations. This includes a new channel in the policy head which learns to

predict the opponent's next-move policy, a new score head which predicts a probability distribution for the score difference at the end of the game, and a new ownership head which predicts which player will own each intersection at the end of the game.

KataGo [6] also adds input features which both provide information to the NN and free up its capacity to learn other things. Both these new inputs and auxiliary targets are mostly domain-specific, only applicable to Go.

Self-Play Dynamics Elf OpenGo [5] observed that AlphaZero can suffer from unhealthy self-play dynamics. Early in training, the model overestimates the white win rate, which causes the black player to resign prematurely, which reduces game diversity in the replay buffer, creating a negative feedback loop. They corrected this by evenly sampling black-win and white-win games.

There are other changes to the self-play dynamics that may remedy a collapse in game diversity or a bias towards one player. For example, Trudeau and Bowling [7] propose Go-Exploit, where self-play trajectories start from "states of interest" (states visited during self-play or MCTS) instead of the starting state. This enables more independent value targets and another source of exploration. Tsai *et al.* [8] builds upon this with Regret Guided Search Control, which uses regret to prioritize which states of interest to start from.

Another modification to self-play is the use of playout cap randomization and policy target pruning in KataGo [6]. Under this approach, only some percentage of turns do full budget MCTS searches. The model is then only trained on the value and policy targets from those turns. This enables more games to be played in the same amount of time, which creates more independent value targets for the value head. The turns without full budget MCTS searches increase exploration, and the policy targets retain their quality because the policy head is only trained on the full budget MCTS turns.

On-Policy Behaviour Willemsen *et al.* [9] observes that AlphaZero behaves more like an on-policy RL algorithm (like SARSA) than an off-policy RL algorithm (like Q-learning) in that the learned policy incorporates the exploration of the behaviour policy used when learning it. Two potential issues with this: (1) the cliff-walking problem [3] where the final policy is too cautious and, (2) unique to self-play, the model only plays against a non-optimal opponent. Willemsen *et al.* [9] found that "the more exploration is added in training, the more the value estimates deviate from the true values for a greedy policy". They remedy this with AlphaZero with greedy backups (A0GB). Instead of using the game outcome z, A0GB follows the MCTS policy greedily until a leaf or terminal state is found. The value of this state is then used as the value target. This allows the value head to converge to the true values for the greedy policy by removing the exploration introduced by the behaviour policy.

Non-Transitive Strategies Some games have a non-transitive strategy space, where strategies can form cycles – strategy A beats strategy B, B beats C, and C beats A. This can be a problem for AlphaZero, because self-play encourages strategies able to defeat the current model. This could cause the model to chase cycles, where it learns a strategy that beats the current model, but is then beaten by a previous strategy. Instead of improving globally, the model oscillates between strategies.

Certain evaluation metrics are unable to capture if a model is improving globally or just learning a locally-strong strategy. For example, self-play Elo (used in ELF OpenGo [5]) gets the win rate between consecutive pairs of models and uses Elo's rating formula [51] to get their rating difference.

ELF OpenGo [5] also uses Nash averaging [52], which captures skill degradations that self-play Elo misses. They observe one segment during training where self-play Elo shows improvement, but Nash averaging shows a decline.

AlphaStar [53] also mentions cycles in the strategy space as a problem in their domain (StarCraft II). They remedy this by replacing self-play with league training, where the current model plays against a set of different players instead of just itself. They also condition the policy head on a strategy vector, which enables the model to learn multiple strategies.

Adversarial Training Wang *et al.* [10] train an adversarial agent that defeats KataGo, a superhuman AlphaZero-style Go player, despite the adversary itself losing to human amateurs. This non-transitivity shows the adversary is exploiting a specific weakness in the victim player rather than simply being a stronger Go player. This suggests AlphaZero-style training leaves gaps in the resulting agent's knowledge that search does not close (their adversary wins 72% of games even when the victim searches 10^7 nodes per move), plausibly related to the exploration collapse and lack of self-play data diversity discussed above.

To train their adversary, they replace self-play with victim-play, where games are played against a fixed victim, and only the adversary's own turns are used as training data. Further, they modify MCTS so that the victim's policy is used during the victim's turn during search, yielding a better response to the victim.

KataGo subsequently added a small amount of hand-curated positions which demonstrate the exploits found by the adversary training to its self-play training. This reduced the adversary's win rate to near zero. However, fine-tuning the adversary recovered a 47% win rate, indicating that such defensive training defends against a specific attacker but can be broken again through more adversarial training.

2.5.3 Analyzing & Improving AlphaZero with Ground Truth

In this thesis, we are able to analyze AlphaZero with ground truth because Chinese Checkers is a strongly solved game. There is previous work analyzing AlphaZero with

ground truth, which we discuss here.

Prasad [12] analyzes AlphaZero on Connect Four, a strongly solved game. They use the solve data to tune hyperparameters and explore how different training strategies (such as replacing the game outcome value target with the value estimate from MCTS) affect the model's performance. Solve data in Connect Four enables a cheap alternative to evaluate AlphaZero-trained players compared to playing against other players to get win rate or Elo.

There is also work analyzing open source AlphaZero-style programs and their behaviour in MCTS, both done in games that are not strongly solved. Haque [13] evaluates Leela Chess Zero (Lc0) using 3, 4, and 5-piece endgame tablebases available in Chess as ground truth. Husna [14] evaluates KataGo using a custom set of solved states generated using a solver as ground truth.

Haque [13] finds that the accuracy of Lc0's raw policy network is correlated with the number of positions at each decision depth more than with the decision depth (depth-to-mate) itself, implying that the NN performs better on decision depths it samples more frequently during training. However, MCTS breaks the correlation between "sample frequency" and error rate, showing that search can solve shallow positions regardless of how often they appeared in training.

Both works [13, 14] observe a search pathology that occurs when the value head is systematically biased. The value estimate term in the UCB formula dominates as visit counts grow, which means more search increases the value head's influence, which does not necessarily lead to better decisions when the value head is biased. When the policy head is correct but the value head is biased, additional search can lead to worse decisions.

Husna [14] expands on this, observing that Go's high branching factor means the search rarely reaches terminal states, making a systematically biased value head more

harmful. They also find that KataGo struggles more with finding the best incentive move than with finding any minimax optimal move, suggesting it optimizes for winning rather than for relative move quality.

Karki [15] evaluates AlphaZero in Chinese Checkers. They find a decrease in policy accuracy on states outside the training set generated by self-play, which adversarial policies were able to exploit. They explore training strategies to encourage robustness to adversarial agents.

Chapter 3

Validating AlphaZero on Chinese Checkers

Analyzing AlphaZero's training dynamics with ground truth requires a working implementation of AlphaZero. In this chapter, we validate that our implementation of AlphaZero is able to learn in two stages. First, we establish the neural network's learning capacity by training it with supervised learning using ground truth data. Second, we validate that the full AlphaZero training procedure generates a model and player that improve over time.

Originally, we intended to extend an AlphaZero implementation developed by a previous Master's student in our lab, Bigyan Karki, who also trained AlphaZero on Chinese Checkers [15]. However, the final version of the code and trained models were lost through administrative error, and we were unable to recreate an operational AlphaZero implementation from the remaining codebase. Therefore, the AlphaZero implementation in this chapter was primarily developed from scratch.

3.1 Establishing Neural Network Learning Capacity

To establish the learning capacity of the neural network at the core of AlphaZero, we create evaluation metrics and a training procedure using our ground truth data.

Ground truth data enables us to cheaply evaluate the quality of the model's value and policy heads compared to, for example, playing games against a baseline player.

3.1.1 Network Evaluation

3.1.1.1 Evaluation Targets

We derive ground truth targets for the value head ($\hat{v}_\theta$) and policy head ($\hat{\pi}_\theta$) from the solve data. For establishing network learning capacity, they are used as labels during supervised learning and as evaluation targets.

Value Head Target The value head target we use as ground truth throughout this thesis is the game-theoretic value of the state given to us by the solve data, v_*. It has the same form as the value target $z \in \{-1, 0, 1\}$ in AlphaZero's training procedure. $v_*(s) = 1$ if the player to move at state s can force a win, $v_*(s) = -1$ if the player to move at state s will lose under optimal play, and $v_*(s) = 0$ if the game is a draw under optimal play.

Policy Head Target The policy head target we use as ground truth throughout this thesis is the uniform optimal policy, π_*,

$$\pi_*(a|s) = \begin{cases} \frac{1}{|A^*(s)|} & \text{if } a \in A^*(s) \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

where $A^*(s)$ is the set of optimal actions for state s,

$$A^*(s) = \{a \in A(s) : -v_*(s_a) = v_*(s)\} \tag{3.2}$$

and s_a is the state reached by taking action a in state s. The negation in $-v_*(s_a)$ is necessary because $v_*(s)$ is from the perspective of the player to move at s. π_* puts an equal probability on all optimal actions and 0 probability on all suboptimal actions.

3.1.1.2 Evaluation Metrics

To evaluate $\hat{v}_\theta$ and $\hat{\pi}_\theta$ against v_* and π_* respectively, we use accuracy.

Value Accuracy Given any value function $\hat{v}$ and a set of states S, value accuracy is the probability that $\hat{v}$ predicts $v_*(s)$ for a state s sampled uniformly from S,

$$\begin{aligned} \text{Acc}(\hat{v} \mid v_*, S) &= \Pr_{s \sim S}\left[\hat{v}(s) = v_*(s)\right] \\ &= \frac{|\{s \in S : \hat{v}(s) = v_*(s)\}|}{|S|} \end{aligned} \tag{3.3}$$

where $s \sim S$ denotes uniform sampling from S.

Our ground truth value function is discrete ($v_* \in \{-1, 0, 1\}$) whereas the learned value head we are evaluating is continuous ($\hat{v}_\theta \in [-1, 1]$). We discretize the value head's output before measuring accuracy.

$$\tilde{v}_\theta = \begin{cases} +1 & \text{if } \hat{v}_\theta \geq 0 \\ -1 & \text{if } \hat{v}_\theta < 0 \end{cases} \tag{3.4}$$

The metric $\text{Acc}(\tilde{v}_\theta \mid v_*, S)$ tells us how often the value head predicts the correct game-theoretic value for states in S. It evaluates how well the value head can estimate the value of a state. Changing what states are in S can tell us what parts of the state space the value head evaluates well – a random sample of states from the state space tells us how well it evaluates states in general, while a sample of states seen during self-play tells us how well it evaluates states it has seen before, or are frequent in self-play.

Policy Accuracy Given any policy $\hat{\pi}$ and a set of states S, policy accuracy is the probability that $\hat{\pi}$'s most preferred action is optimal ($\in A^*(s)$) for a state sampled uniformly from S,

$$\begin{aligned}\mathrm{Acc}(\hat{\pi} \mid \pi_*, S) &= \Pr_{s \sim S}\left[\operatorname*{argmax}_{a \in A(s)} \hat{\pi}(a|s) \in A^*(s)\right] \\ &= \frac{\left|\left\{s \in S : \operatorname{argmax}_{a \in A(s)} \hat{\pi}(a|s) \in A^*(s)\right\}\right|}{|S|}\end{aligned} \tag{3.5}$$

where argmax selects the action the policy assigns the most weight; ties are broken by randomly selecting one of the tied actions.

3.1.1.3 Standardizing State Sets

When performing evaluations, we need to consider the structure of the set of states we are evaluating $\hat{v}$ or $\hat{\pi}$ on. The distribution of states in a set can inflate or deflate accuracy independently of the model's ability, making performance comparisons across different sets (for example, between states seen during self-play and states sampled from the full state space) difficult. We modify each set so that differences in accuracy better reflect differences in model performance rather than set composition.

Value Head Filter Before evaluating $\hat{v}$ on a set of states, we balance the dataset so it has an equal number of winning states $v_*(s) = 1$ and losing states $v_*(s) = -1$. We also remove the draw states $v_*(s) = 0$. On an unbalanced set, a model biased towards the majority class can score well without distinguishing between states. Draw states are removed because the discretized value head cannot predict them. Every draw state is necessarily scored incorrect, so a set with many draws lowers accuracy regardless of the model's quality. For example, a discretized model like $\tilde{v}_\theta$ would be scored at least 50% inaccurate on a dataset with 50% draw states, with no such penalty for a set without draw states. Balancing and removing draw states keeps scores comparable across sets.

We define this filter operation:

$$\Phi_v(S) = \text{Balance}(\{s \in S : v_*(s) \neq 0\}) \tag{3.6}$$

where Balance$(\cdot)$ removes a uniformly random subset of the majority class until the number of winning and losing states are equal. We write $S^v = \Phi_v(S)$ as shorthand, so value head accuracy on a dataset S is reported as $\text{Acc}(\tilde{v}_\theta \mid v_*, S^v)$.

Policy Head Filter We also modify datasets prior to measuring policy head accuracy by removing trivial states. Trivial states are states where every legal action is optimal, $A^*(s) = A(s)$. Just as draw states deflate value accuracy, a set with many trivial states inflates policy accuracy regardless of the model's quality. Removing trivial states keeps scores comparable across sets.

$$\Phi_\pi(S) = \{s \in S : A^*(s) \neq A(s)\} \tag{3.7}$$

We write $S^\pi = \Phi_\pi(S)$ as shorthand, so policy head accuracy on a dataset S is reported as $\text{Acc}(\hat{\pi}_\theta \mid \pi_*, S^\pi)$.

3.1.2 Network Training

In this section, we describe the architecture of the neural network and the training procedure we use to establish its learning capacity.

3.1.2.1 Network Architecture

The model's architecture is shown in Figure 3.1. Hyperparameters such as batch size and learning rate for the network and its training are reported in Table B.1.

3.1.2.2 Training Procedure

This section describes the training procedure we use to establish the neural network's learning capacity. We train the model with supervised learning on ground truth data,

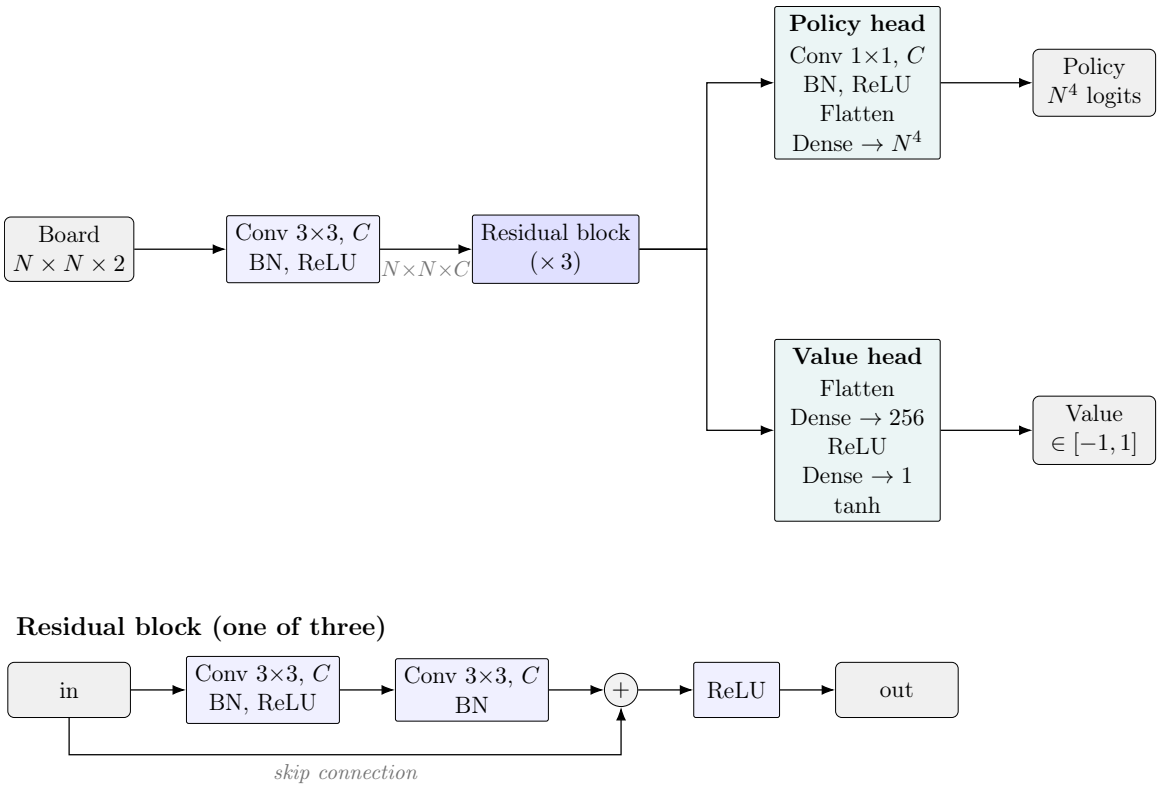


Figure 3.1: The AlphaZero network: a shared residual convolutional trunk feeding separate policy and value heads. N is the board size and C the number of convolutional filters (`num_filters`); the trunk stacks three residual blocks, one of which is expanded below. All convolutions are 3×3 except the policy head's 1×1 projection.

and evaluate its performance on both train and test sets. If the model's value and policy accuracy increases on both sets, we conclude that the model has the capacity to learn.

First, we create a training and evaluation dataset (S_{train} and S_{eval}) consisting of tuples $(s, v_*(s), \pi_*(s))$, where s is the input and $v_*(s)$ and $\pi_*(s)$ are the value and policy head targets, respectively. Both are created by uniformly sampling states from the state space. We balance the number of winning and losing states in S_{train} so we don't train a model on an imbalanced dataset.

During training, we evaluate the model's performance on both S_{train} and S_{eval} at regular intervals. We evaluate the value head with $\text{Acc}(\tilde{v}_\theta \mid v_*, S^v_{\text{train}})$ and $\text{Acc}(\tilde{v}_\theta \mid v_*, S^v_{\text{eval}})$. We evaluate the policy head with $\text{Acc}(\hat{\pi}_\theta \mid \pi_*, S^\pi_{\text{train}})$ and $\text{Acc}(\hat{\pi}_\theta \mid \pi_*, S^\pi_{\text{eval}})$. By evaluating on both S_{train} and S_{eval}, we can determine whether the model is overfitting, underfitting, or generalizing well.

We do this for two different board sizes, 16-3 and 25-6. In addition, we repeat this training procedure for different dataset sizes (1,024, 8,192, 65,536, 524,288) to see how they affect the final accuracy of the model after 10 epochs of training.

3.1.3 Results

If our model has the capacity to learn, we expect to see an increase in the accuracy of the model on both the training and evaluation datasets. In the 16-3 setting, both the value and policy heads converge to 94% accuracy on the S_{eval} states. In the 25-6 setting, the value head converges to 92% accuracy and the policy head converges to 85% accuracy. In both settings, the value head converges to its final accuracy faster than the policy head. We also see that the model's accuracy on S_{train} is higher than on S_{eval}, which is expected because the model has seen these states during training. The value head and policy head accuracy curves over training (on 65,536 states) are

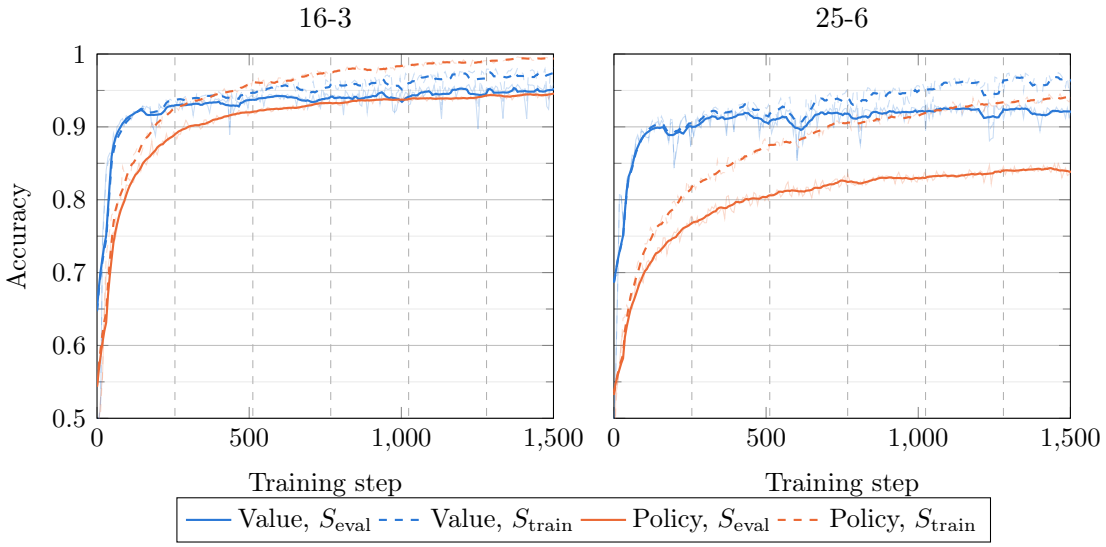


Figure 3.2: Supervised learning training curves on 16-3 (left) and 25-6 (right): value and policy head accuracy on the S_{train} and S_{eval} state sets over ten epochs of training on 65,536 ground truth states with a three-block network. Faint vertical lines mark epoch boundaries. Curves are smoothed with a centered moving average over 11 evaluations; the unsmoothed curves are shown at low opacity.

shown in Figure 3.2. These results confirm that our model has the capacity to learn the ground truth value and policy targets.

The final accuracy of both the value and policy heads increases with dataset size. The policy head's accuracy is more sensitive to dataset size than the value head's accuracy. In both settings, the value head only needs 8,192 states to achieve above 90% accuracy on S_{eval}, while the policy head needs 65,536 states (in 16-3) and 524,288 states (in 25-6). This information is useful for calibrating expectations when training with AlphaZero. The policy head needs more training data to learn than the value head, and we likely need more states than we trained on in this setting because AlphaZero's training targets are noisier than the ground truth targets. These results are shown in Figure 3.3.

It is useful to consider the percentage of the entire state space that the training dataset represents. These numbers are reported in Table 3.1. In the 25-6 setting, all

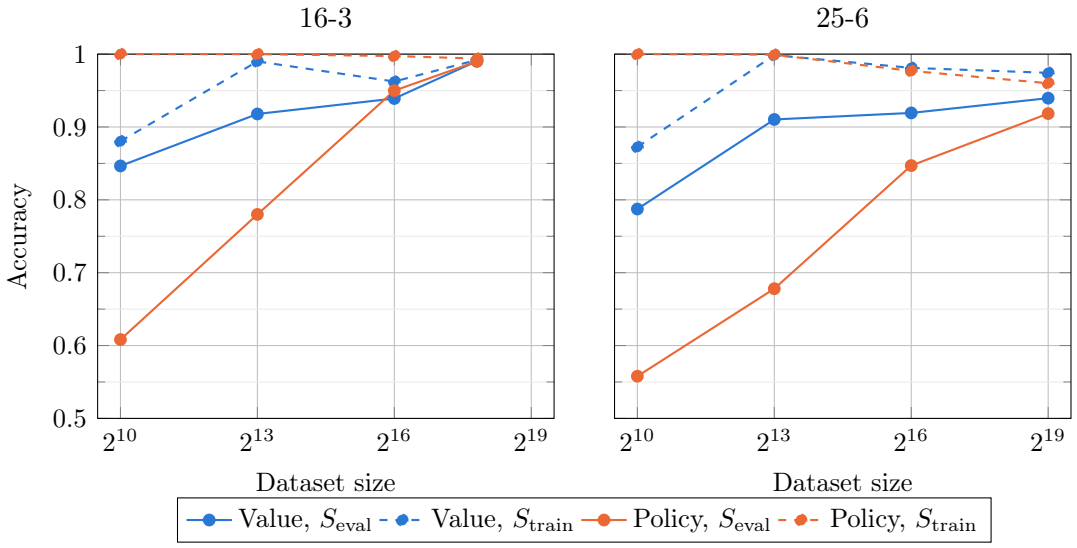


Figure 3.3: Final accuracy after ten epochs of supervised learning versus training set size, on 16-3 (left) and 25-6 (right), with a three-block network. Accuracy is measured on the S_{train} and S_{eval} state sets.

Dataset size	16-3	25-6
1,024	0.32%	1.07×10^{-5} %
8,192	2.56%	8.52×10^{-5} %
65,536	20.5%	6.82×10^{-4} %
524,288	100%	5.46×10^{-3} %

Table 3.1: Each training dataset size as a percentage of the setting's state space. The 16-3 setting has 320,320 unique states; the 25-6 setting has 9,610,154,400.

dataset size settings represent a negligible fraction of the entire state space, so accuracy on S_{eval} is a good indicator of generalization. In the 16-3 setting, however, there is a good chance of overlap between the randomly sampled S_{train} and S_{eval} datasets. The largest training set covers the entire state space, so S_{eval} contains no unseen states and accuracy there measures memorization rather than generalization. It makes sense that the model converges to approximately 100% in this setting.

3.2 Establishing AlphaZero Learning Ability

To establish that our AlphaZero implementation is able to learn, we evaluate the model's accuracy on the ground truth targets and the AlphaZero player's win rate against a baseline player over training iterations. In addition, we also evaluate the accuracy of the training data generated by self-play. This helps us with our goal of understanding AlphaZero's training dynamics.

3.2.1 AlphaZero Evaluation

To evaluate the strength of the AlphaZero player, we define a baseline player for it to play against. We also define evaluation metrics to measure the player's strength and the training data accuracy.

3.2.1.1 Player Evaluation

Baseline Player The baseline player is a standard MCTS player. It uses UCT as its selection policy. For state evaluations, it performs a rollout, then evaluates the resulting state using a heuristic evaluation function described below. The rollout policy is to choose the action that moves a piece the furthest forwards, with an ϵ probability of choosing a random forward action instead.

The evaluation heuristic, DistEval, values states where the player's pieces are closer to their goal area than the opponent's are to theirs, with a small bonus when it is the

player's turn. To evaluate a state, it calculates the distance of each player's pieces to their respective goal areas and sums these distances for each player. A state's DistEval value is the difference between these sums, along with a turn bonus for the player to move. Formally, when evaluating state s for player i with opponent j,

$$\text{DistEval}_i(s) = \sum_{p \in P_j(s)} d(p, G_j) - \sum_{p \in P_i(s)} d(p, G_i) + b_i(s) \tag{3.8}$$

where $P_i(s)$ is the set of board positions occupied by player i's pieces in state s, G_i is player i's goal area, $d(p, G)$ is the distance from position p to goal area G, and $b_i(s)$ is the turn bonus, equal to 1 if player i is to move at s and 0 otherwise.

The hyperparameters for the baseline player are reported in Table B.4.

Player Evaluation Metrics The primary evaluation metric is win rate against the baseline player. To compute this, the player being evaluated and the baseline player play a number of games against each other. The win rate is the number of games won by the player being evaluated divided by the total number of games played. We also measure the loss and draw rates.

In Chinese Checkers, the first player has an advantage. Therefore, we separate the win, loss, and draw rates by whether the player starts as the first (P1) or second player (P2).

3.2.1.2 Training Data Evaluation

We also measure the accuracy of the training data generated by self-play (the targets in SP) that the neural network is trained on. While this doesn't establish the player's learning, it does help us investigate the training dynamics of AlphaZero. For example, we can monitor if the training data is improving or degrading over time, and if the model is learning from it.

One consideration when evaluating the accuracy of the self-play tuples is how to handle tuples where $z = 0$ (i.e. $(s, \pi, 0)$). These tuples are nearly always evaluated as inaccurate because the solve data has relatively few draw states (16-3: 0.0%; 25-6: 0.5%; more details in Table A.1). If half of self-play games ended in draws, the training data would be evaluated as at least 50% inaccurate, even before the other tuples are evaluated. Due to this, we remove tuples where $z = 0$ before evaluating the training data value accuracy. This resulting set of tuples we denote as SP^v. We look at the win/loss/draw percentages of the self-play tuples separately.

On the policy side, we remove tuples with trivial states (states s where $A^*(s) = A(s)$, or $\rho(s) = 1$), because they do not meaningfully evaluate the policies in SP. The resulting set of tuples we denote as SP^π. We look at the proportion of trivial states in the self-play tuples separately.

Another consideration is state uniqueness. The same state can be seen multiple times during self-play. Further, the same state can have different targets associated with it. For example, $(s, \pi, 1)$ and $(s, \pi', -1)$ can both be in SP. This motivates a new accuracy definition, because $\text{Acc}(\hat{v} \mid v_*, S)$ assumes the model being evaluated has a single prediction for each state.

We therefore define an accuracy metric for a list of samples rather than a model. A sample is accurate if its target matches the ground truth at its state. We largely reuse the metric concepts developed in Section 3.1.1.2. For value targets:

$$\begin{aligned} \text{Acc}(\text{SP} \mid v_*) &= \Pr_{(s,\pi,z)\sim\text{SP}}[z = v_*(s)] \\ &= \frac{|\{(s, \pi, z) \in \text{SP} : z = v_*(s)\}|}{|\text{SP}|} \end{aligned} \tag{3.9}$$

For policy targets:

$$\begin{aligned}\text{Acc}(\text{SP} \mid \pi_*) &= \Pr_{(s,\pi,z)\sim\text{SP}}\left[\operatorname*{argmax}_{a\in A(s)} \pi(a|s) \in A^*(s)\right] \\ &= \frac{\left|\left\{(s,\pi,z) \in \text{SP} : \operatorname{argmax}_{a\in A(s)} \pi(a|s) \in A^*(s)\right\}\right|}{|\text{SP}|}\end{aligned} \tag{3.10}$$

3.2.2 AlphaZero Training

To validate the learning ability of our AlphaZero implementation, we train a model with AlphaZero on the 16-3 and 25-6 board sizes, following the training procedure described in Section 2.3 (AlphaZero). We then evaluate the model's accuracy on the ground truth targets and the AlphaZero player's win rate against the baseline player over training iterations. We also evaluate the accuracy of the training data generated by self-play (SP), the samples the model is trained on.

When evaluating the model's accuracy, we evaluate on four datasets. The first is a random set of states sampled from the state space, S_{random}. The second is a set of states the model has seen during self-play (in SP), S_{seen}. The third and fourth are neighboring datasets, which are sets of states one move away and two moves away from the states in S_{seen}, $S_{\text{neighbor1}}$ and $S_{\text{neighbor2}}$. This gives us an idea of how well the model has generalized to states it has not seen during self-play. Value accuracy is calculated with $\text{Acc}(\tilde{v}_\theta \mid v_*, S^v)$ and policy accuracy with $\text{Acc}(\hat{\pi}_\theta \mid \pi_*, S^\pi)$, where S is one of the four datasets.

Training data accuracy is evaluated on the self-play tuples generated at each iteration i, SP_i. Value accuracy is calculated with $\text{Acc}(\text{SP}_i^v \mid v_*)$ and policy accuracy with $\text{Acc}(\text{SP}_i^\pi \mid \pi_*)$.

We train for 25 seeds on each board size, then report the mean and 95% confidence interval of the metrics described above. Player evaluations were run for 15 of those seeds. We report hyperparameters used in Appendix B.2.

3.2.3 Results

If our AlphaZero implementation is working correctly, we expect to see an increase in the accuracy of the model, and an increase in the AlphaZero player's win rate against the baseline player. In 16-3, the model's value head accuracies on all four datasets converge to between 88% and 91%, and the policy head accuracy converges to between 88% and 94%. In 25-6, the model's value head accuracies on all four datasets converge to between 84% and 86%, and the policy head accuracy converges to between 83% and 86%. In all cases, the model's accuracies on all four datasets closely track each other. Further, the model's accuracy quickly converges to its final value within 10 iterations, though it continues to slowly rise afterwards. These results, the model and training data accuracy curves over training, are shown in Figure 3.4.

In 16-3, the player's win rate against the baseline player converges to approximately 98% as P1, and to about 82% as P2. The P2 win rate stays noticeably more variable across seeds than the P1 win rate through iteration 50. In 25-6, the win rate converges to about 90% as P1 and about 82% as P2, with the P2 win rate rising more slowly. These results, the player's win/loss/draw rates against the baseline player, are shown in Figure 3.5.

Our trained model achieves high accuracy and as a player performs well against the baseline player. These results validate that our AlphaZero implementation is able to learn.

For the policy head, the model's accuracy closely tracks the training data accuracy. For the value head, however, the model's accuracy is consistently higher than the training data accuracy. This has been a robust and surprising result across our AlphaZero implementation and experiments.

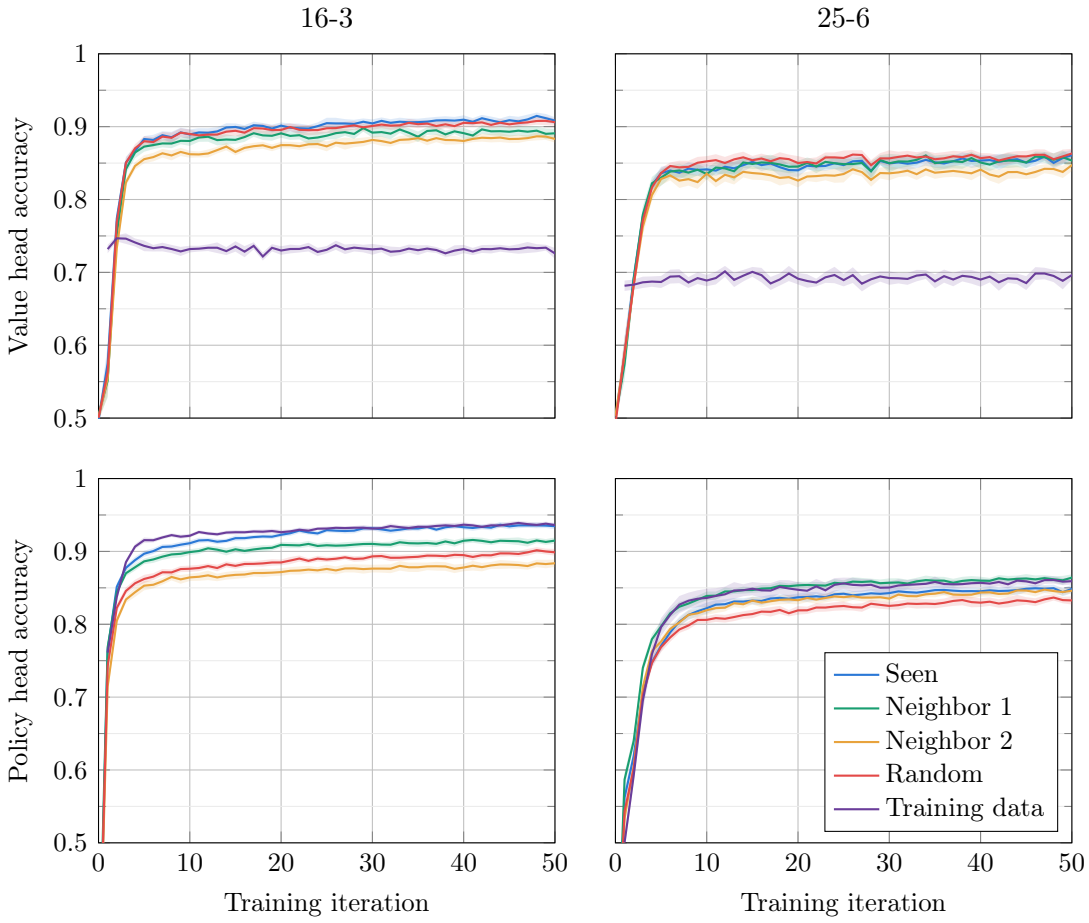


Figure 3.4: Model accuracy on ground truth over training, across evaluation datasets. Columns show the 16-3 (left) and 25-6 (right) boards; rows show the value (top) and policy (bottom) heads. All panels share the same accuracy axis.

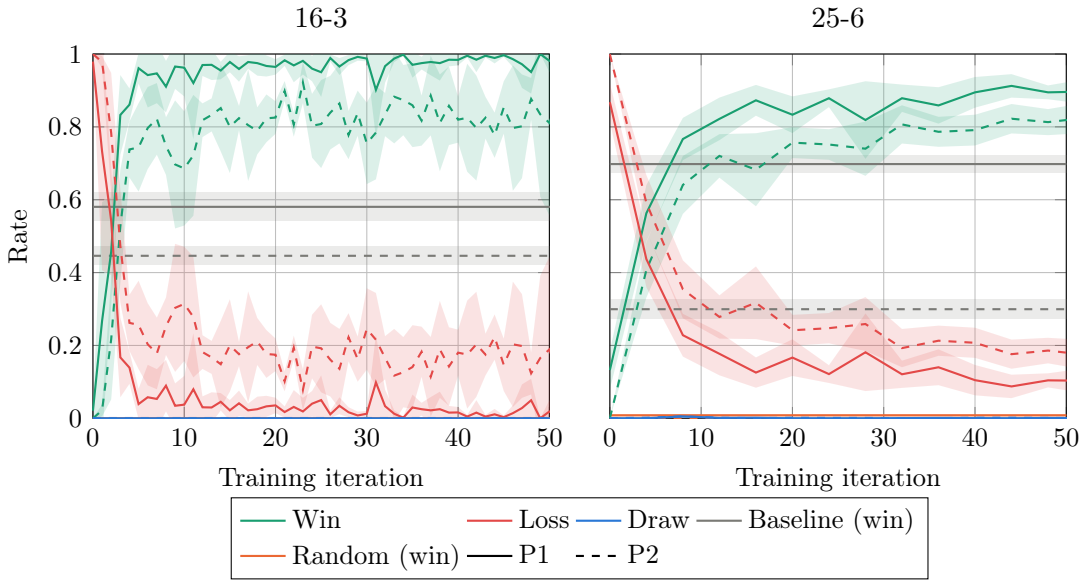


Figure 3.5: Win, loss, and draw rates of the AlphaZero player against the baseline over training, on 16-3 (left) and 25-6 (right). Colour indicates the rate; solid and dashed lines show the AlphaZero player as first and second player. Horizontal reference lines show the win rates of the baseline against itself and of the random player against the baseline. Shaded bands show 95% confidence intervals.

3.2.4 Discussion and Related Work

That the value head accuracy is consistently higher than the training data accuracy is a surprising result. How can the value head learn to be more accurate than its training data? There is a large body of work on learning from noisy labels, from which we can draw from to investigate this result. For example, we can check if it learns to approximate the Bayes optimal classifier [54]. For a state s, the Bayes optimal classifier predicts the value that occurs most often in the training data at that state, $\operatorname{argmax}_{v \in \{-1,0,1\}} \Pr_{(\tilde{s},\pi,z)\sim \text{SP}}[z = v \mid \tilde{s} = s]$. The classifier is only defined on the states that appear in SP. If the model is about as accurate as the Bayes optimal classifier when evaluated against ground truth, then it is just learning to approximate the training data distribution. If the model is more accurate than the Bayes optimal classifier, then its higher accuracy must be from some other structure(s) in the data.

For example, perhaps states from later stages of the game are more accurate, and the model is able to generalize from these states. We did some initial investigations into this direction and found that the training data accuracy and the model's value head accuracy are higher on end game states.

We also discuss whether there is a metric or ground truth target that better reflects the model's quality in Appendix B.1. Targets and metrics that account for discounting and utilize DTW information are one promising direction.

Chapter 4

AlphaZero with Ground Truth

4.1 Motivation & Background

Ground truth allows us to investigate AlphaZero's learning process both through passive observation, such as by evaluating the trained model's accuracy or the accuracy of the training data against ground truth, or through active intervention. For example, we can investigate how a more accurate self-play training dataset would affect the model.

This experiment was run by Bigyan Karki, a former member of our lab who also conducted research with AlphaZero on Chinese Checkers, during his Master's. Karki *et al.* [16] "replace[d] the outcome of the game (z) in each of the training tuples [...] with the ground truth value of the game". The expected result of increased training data accuracy was that the player would improve. However, the resulting player was "far worse than our original trained player without ground truth" [16].

This is a surprising result. We decided to re-run this experiment to determine whether the result replicates and, if so, why.

4.2 Method

In AlphaZero, self-play at each iteration i is used to generate a list of tuples $\{(s, \pi, z)\}$, SP_i. The tuples are then used to train the model, with z as the target for the value head and π as the target for the policy head. In this experiment, we replace z with the ground truth value of the state, $v_*(s)$, and π with the ground truth policy, $\pi_*(s)$. This new list of tuples is denoted $\text{SP}_{*,i} = \{(s, \pi_*(s), v_*(s))\}$. The model is then trained on the targets in $\text{SP}_{*,i}$ instead of SP_i. The rest of the training process is unchanged from the standard AlphaZero training process described in Section 3.2.2 (AlphaZero Training).

While Section 3.2.2 evaluates model accuracy on the four datasets S_{seen}, $S_{\text{neighbor1}}$, $S_{\text{neighbor2}}$, and S_{random}, we report accuracy on only S_{seen} and S_{random} in Figure 4.1.

4.3 Results

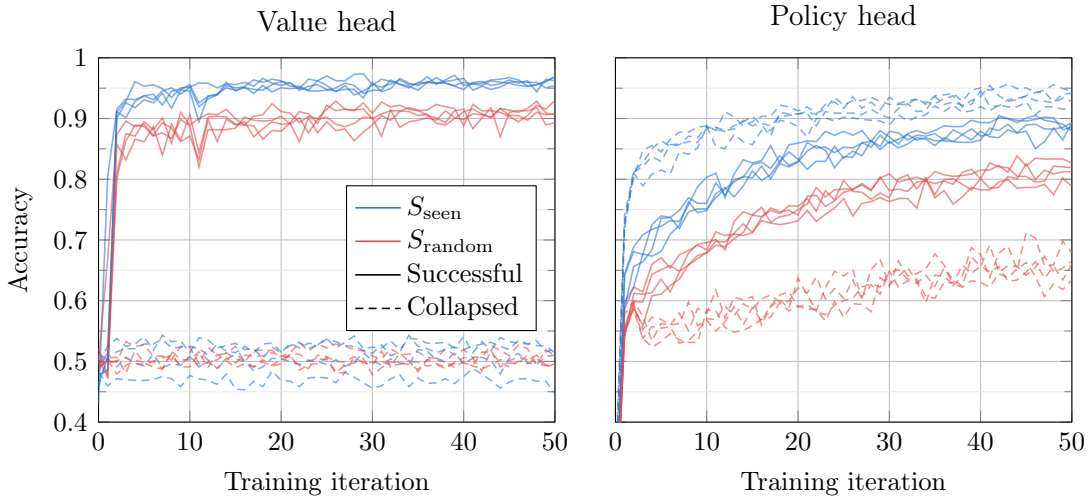


Figure 4.1: Model accuracy on ground truth over training when trained on ground-truth targets (SP_*), with each of the 8 trials drawn separately. $\hat{v}_\theta$ and $\hat{\pi}_\theta$ are evaluated on the S_{seen} and S_{random} datasets. Solid lines mark the four runs whose value head learned successfully, and dashed lines the four whose value head collapsed; the policy head uses the same per-trial styles. The value head (left) and policy head (right) share the accuracy axis.

The seeds are bimodal – some learn successfully while others fail to learn. The four successful runs have a value accuracy of about 90% on S_{random} while the four unsuccessful runs have a value accuracy of about 50%. These results are shown in Figure 4.1. We show the accuracy curves of each seed instead of the mean and 95% confidence interval over all seeds because of the bimodality.

This is an unexpected result. We expected that the model would either learn successfully because it is trained on ground truth, or that we would replicate the results from Karki *et al.* [16] and see the model fail to learn. Instead, we see that the model sometimes learns successfully and sometimes fails to learn. To investigate why some runs fail while others succeed, we looked at additional artifacts generated by the training process. We hypothesized at this time that the unsuccessful runs could be failing because of some self-play issue (e.g., exploration collapse, feedback loops), misalignment between the value and policy targets (exacerbated by the shared backbone), or some other factor.

4.4 Investigation

4.4.1 Value Head Prediction Distribution

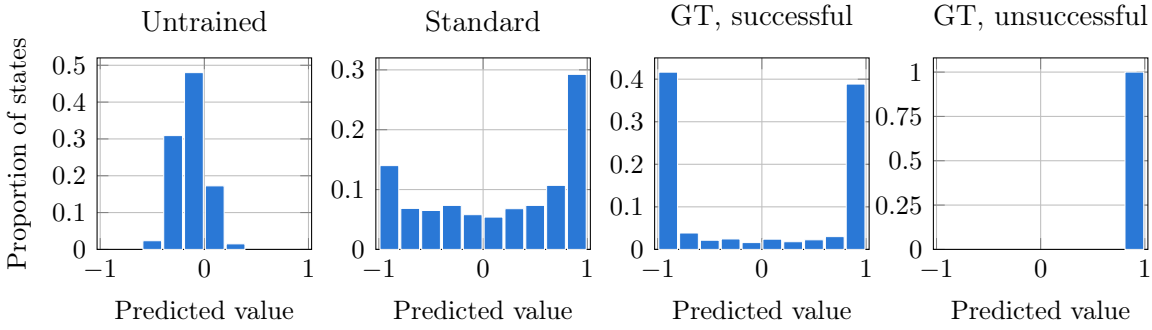


Figure 4.2: Distribution of the value head's predictions on 2000 states from S^{v}_{random}, binned into 10 equal bins over $[-1, 1]$ and shown as the proportion of states in each bin. The y axes differ across panels. From left to right: an untrained model, a normally trained model, a successful ground-truth run, and an unsuccessful ground-truth run.

We plot a prediction distribution histogram of the final model's value head on S^v_{random}. To do this, we count how many predictions fall into the bins [-1, -0.8), [-0.8, -0.6), ..., [0.8, 1] and plot the resulting distribution. We do this on four models. (1) An untrained model, (2) a model from a normal AlphaZero run (described in Section 3.2.2), (3) a model from a successful "ground truth" run, and (4) a model from an unsuccessful "ground truth" run. We plot the distributions of a single model in each setting rather than an average or combined distribution. The results are shown in Figure 4.2.

The untrained model has a normal prediction distribution. The successful runs have a U-shaped distribution, with the ground truth replacement setting being more extreme. The model from the unsuccessful run only predicts 1. This suggests that the low accuracy of the unsuccessful runs in Figure 4.1 is due to the value head collapsing to always predict a single value (1 or -1) for all states. Further, we see this collapse early in training. Three of the runs collapse to this prediction distribution at iteration 3, the other at iteration 6.

4.4.2 Training Data Imbalance

The above results suggest that the balance of the training data in early iterations is a factor in why some runs collapse. We define p_{maj} as the proportion of the majority class in the training data, counting only targets in $\{-1, +1\}$ (targets of 0 are excluded as they don't contribute to the imbalance). A balanced dataset has $p_{\text{maj}} = 0.5$, while a completely imbalanced dataset has $p_{\text{maj}} = 1$.

For all runs, the training data starts imbalanced at about $p_{\text{maj}} = 0.65$. Then by iteration 5, p_{maj} converges to about 0.55 for the rest of the training. Therefore, the prediction distribution collapse seen in the value head happens when $p_{\text{maj}} \approx 0.60$. The imbalance is similar across both the successful and unsuccessful runs. This similarity

suggests that the imbalance is not the cause of the collapse.

For context, regular AlphaZero training creates a dataset with $p_{\text{maj}} \approx 0.5$. This is because z in the self-play tuples (s, π, z) is derived from the game outcome. A training tuple is generated for each turn in the game, and the game alternates between players. If the game is won by P1, then all of P1's states will have $z = 1$ and all of P2's states will have $z = -1$. Games with even lengths will have an equal number of states for each player, while games with odd lengths will have one more state for the winning player. Only odd games below 12 turns would replicate the $p_{\text{maj}} \approx 0.55$ imbalance seen in the ground truth runs (a 13-turn game gives $p_{\text{maj}} \approx 0.54$).

To isolate the effect of the imbalance from any self-play effects, we train a model on a dataset that mimics the imbalance of the early iterations. We train a model on the S_{train} dataset from the experiment described in Section 3.1.2.2 but with subsampling performed to create a majority-class fraction of $p_{\text{maj}} = 0.6$. With 30 seeds, we see 15 runs collapse and 15 runs learn successfully ($50\% \pm 17\%$ 95% CI). This suggests that the bimodal result on the ground truth runs is due to this bimodal property of the neural network, enabled by the imbalance in the training data.

4.4.3 Balancing the Training Data

The above suggests that training data imbalance is the cause of the collapsed runs. To confirm this, we repeat the ground truth experiment but balance the training data by subsampling the majority class. In AlphaZero, the self-play tuples in SP_i generated at iteration i are put into a replay buffer. To train the model at iteration i, a batch of tuples is sampled from the replay buffer. To balance the model's training data, we subsample the majority class in the dataset sampled from the replay buffer. This ensures that the model is trained on a balanced dataset at each iteration.

The value head targets in the dataset are in $\{-1, 0, 1\}$. To balance the dataset, we

only balance the number of samples with value head targets of -1 and 1. Samples with value head targets of 0 are ignored as they don't contribute to the imbalance.

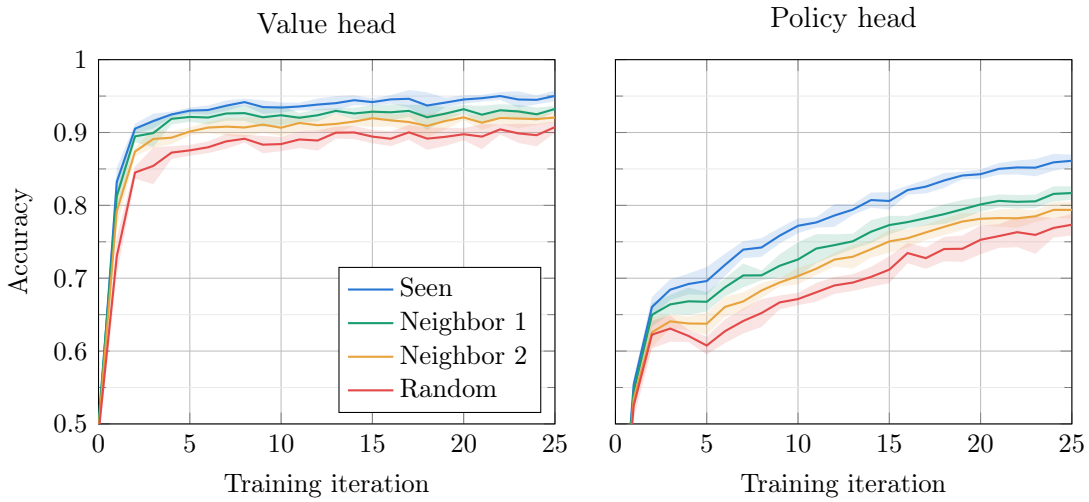


Figure 4.3: Model accuracy on ground truth over training when trained on balanced ground-truth targets, across evaluation datasets. The value head (left) and policy head (right) share the accuracy axis, showing the mean and 95% confidence interval over 8 runs.

The results of this modification are shown in Figure 4.3. All runs learn successfully, confirming that the imbalance in the training data is the cause of the collapsed runs. We see that the value head converges to approximately 95% on seen states and 90% on random states. Similarly, we see the policy head learning. It reaches 85% accuracy on seen states and 77% accuracy on random states. The policy head does not appear to have converged, so higher accuracy is likely attainable with longer training. We also see noticeable banding in the accuracy curves, with the model's accuracy on seen states being higher than on neighboring states, which is higher than on random states.

4.5 Discussion & Related Work

4.5.1 Majority-Class Collapse Mitigations

Majority-class collapse is a failure mode in machine learning where an imbalanced dataset causes a model to constantly predict the dominant class. Instead of learning to distinguish between classes, the imbalance allows this constant prediction strategy to reduce the loss. This is a known problem in machine learning with a large body of literature attempting to address it [55]. The problem is commonly seen in classification tasks with imbalanced datasets, mostly some variant of anomaly detection (e.g. fraud detection, spam detection, object detection, medical diagnosis, named entity recognition, etc.) or tasks with skewed distributions (e.g. sentiment analysis, text classification, image classification, etc.).

Strategies to mitigate majority-class collapse tend to focus on either addressing the issue at the data level (balancing the dataset) or at the loss level (changing what the model is optimizing for) [55]. We discuss these strategies and explore other mitigations specific to our training setup.

4.5.1.1 Balancing the Dataset

The most common strategy to mitigate majority-class collapse is to balance the dataset. There are two main ways to do this: subsampling the majority class or upsampling the minority class by generating synthetic data.

Subsampling the majority class involves removing samples from the majority class until the dataset is balanced. This is what we did to fix the majority-class collapse in our experiments. However, this strategy discards data. In extreme cases, like when the minority class is only a few percent of the dataset, the reduced dataset may be too small to train a model effectively. This would make the strategy infeasible [56].

Generating synthetic data involves creating new samples for the minority class

to balance the dataset. This is commonly done with SMOTE (Synthetic Minority Oversampling Technique) [57] which generates synthetic samples by interpolating between existing samples in the minority class. However, this is not applicable to our case because the input is a board state, and interpolating between two board states does not yield a valid board state.

In our case, we can take advantage of board symmetries to generate synthetic data. Chinese Checkers has two types of symmetry. The first is right-left symmetry, where the board is flipped horizontally along the vertical axis. The second is P1-P2 symmetry, where the board is flipped vertically along the horizontal axis and the players are swapped [11]. The P1-P2 symmetrical board state is also right-left symmetrical, so we can generate 3 synthetic samples for each sample in our dataset. However, our model is player-agnostic, so one of the symmetries is already accounted for by the input representation. Therefore, we can at most double the number of samples in the minority class.

4.5.1.2 Modifying the Loss Function

Standard losses tend to reward an increase in accuracy on the training set. When one class dominates, an easy way for the model to lower loss is to predict the majority class. However, instead of fixing the dataset imbalance, we can modify the loss function to make this naive strategy infeasible.

A common approach is class weighting. This adds weights to each sample so the loss and resulting gradient updates are balanced between classes. This means that minority class misclassifications are penalized heavily, making the "just predict the majority class" strategy more expensive.

For example, if the dataset has 6 winning samples and 4 losing samples, we can weight the loss of each sample according to $w_i = \frac{N}{K \cdot N_i}$, where N is the total number of samples, K is the number of classes (2), and N_i is the number of samples in class i.

In this case, we would weight winning samples with $\frac{10}{2 \cdot 6}$ and losing samples with $\frac{10}{2 \cdot 4}$. This ensures that the total loss of the winning samples is equivalent to the total loss of the losing samples (both are 5), which is the same weight for both classes as if the dataset was balanced and no weighting was applied.

4.5.1.3 Thesis-Specific Mitigation Strategies

Our investigations above can give us clues into mitigation strategies that would work in our specific training setup. One important observation is that we use MSE loss on the value head. MSE loss is minimized when the model predicts the mean of the targets. Therefore, if the model is trained on a dataset with 60% winning samples and 40% losing samples, the model should converge to predicting 0.2 on average. That we observe the model always predicting 1 suggests that there is instability in the value head's learning. Like majority-class collapse, addressing instability in the training of neural networks is also a well-studied problem in the literature with many approaches [58]. To name some: lower learning rate, gradient clipping [59], fine-tuning the optimizer, weight initialization [60], normalization layers [61], residual connections [62], and architecture updates.

On architecture updates, the architecture of the original AlphaZero value head starts with a convolution of 1 filter, then is followed by 2 fully connected layers [4, 39]. Our trunk outputs $256 \times N \times N$ activations for an $N \times N$ board. The original's convolution reduces this to $1 \times N \times N$ before flattening, so its first fully connected layer takes N^2 input features. Our value head skips this convolution and flattens the trunk's output directly, so our first fully connected layer takes $256N^2$ input features, and thus has far more weights. Since larger, more heavily parameterized layers can be harder to train stably, reintroducing this convolution is a plausible way to stabilize the value head.

Finally, that the value head collapse only happens in the first few iterations suggests

that we could simply add a retry mechanism to our training process.

Chapter 5

AlphaZero with TD(λ)

5.1 Motivation

Our ground truth experiments (Chapter 4) revealed that alternative targets can create imbalanced datasets, which can cause value head instability and majority-class collapse in the network. It also suggested potential remedies to this problem. Therefore, we decided to investigate alternative targets to AlphaZero's default Monte Carlo (MC) targets, enabled by our new ability to mitigate this training issue.

We decided to investigate TD(λ) targets. In this chapter, we define the TD(λ) setting in AlphaZero and discuss the balance of its target distribution. We then describe two training collapse mitigation strategies we explored to enable TD(λ) training. Finally, we present our results and potential extensions to TD(λ) training in AlphaZero.

5.2 The TD(λ) Setting

5.2.1 TD(λ) Targets

By default, AlphaZero uses Monte Carlo (MC) targets (described in Section 2.1.2),

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1} \tag{5.1}$$

where r_{t+k+1} is the reward at time $t+k+1$, γ is the discount factor, and T is the time step of the end of the episode. G_t is the experienced return at time t. In Chinese Checkers, the only reward is at the end of the game, we do not use discounting, and it is a two-player adversarial game. The MC target for all states therefore reduces to the game outcome from the perspective of the player to move at s_t.

$$G_t = \begin{cases} +1 & \text{if the player to move at } s_t \text{ wins} \\ -1 & \text{if the player to move at } s_t \text{ loses} \\ 0 & \text{if the game is a draw} \end{cases} \tag{5.2}$$

This leads to a balanced training dataset, as discussed in Section 4.4.2.

To define TD(λ) targets, we first recall the n-step TD target definition from Section 2.1.3.

$$G_{t:t+n} = \sum_{k=0}^{n-1} \gamma^k r_{t+k+1} + \gamma^n \hat{v}(s_{t+n}) \tag{5.3}$$

where $\hat{v}(s_{t+n})$ is the value estimate of the state at time $t+n$.

As with MC targets, this simplifies to

$$G_{t:t+n} = (-1)^n \, \hat{v}(s_{t+n}) \tag{5.4}$$

where $(-1)^n$ ensures $G_{t:t+n}$ is in the perspective of the player to move at s_t.

The TD(λ) target is a weighted sum of all n-step targets,

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{T-t-1} G_t \tag{5.5}$$

where $\lambda \in [0,1]$ is a hyperparameter that controls the weighting of the different n-step targets. When $\lambda = 0$, the TD(λ) target G_t^λ reduces to the TD(0) target, $-\hat{v}(s_{t+1})$. When $\lambda = 1$, G_t^λ reduces to the MC target described in Equation (5.2). Values of λ between 0 and 1 interpolate between these two targets.

5.2.2 TD(λ) Targets in AlphaZero

In the default AlphaZero setting, self-play generates a list of tuples (s_t, π_t, G_t), where G_t is the MC target. We call the list of tuples generated at iteration i SP_i. In our TD(λ) setting, we replace G_t with G_t^λ, creating a list of tuples $(s_t, \pi_t, G_t^\lambda)$ at iteration i, which we call SP_i^λ. For all tuples in SP_i^λ, G_t^λ is calculated using $\hat{v}_{\theta_{i-1}}$ (the value head of the model generated at iteration $i-1$).

The rest of the training and evaluation procedure is the same as in the default AlphaZero setting described in Section 3.2.2 (AlphaZero Training), except that the model is trained on the targets in SP_i^λ instead of SP_i, and training data value accuracy is calculated using the targets in SP_i^λ instead of SP_i.

5.2.3 TD(λ) Target Distribution

TD(λ) is a weighted sum of n-step TD targets, which, in this setting, are all the model's own predictions. The only exception is the terminal return G_t, which is the game outcome. The weight on the terminal return G_t is λ^{T-t-1}, which decays geometrically by the number of steps to the end of the game. For $\lambda = 0.5$, the weight on G_t is 1 for s_{T-1} (the final non-terminal state), 0.5 for s_{T-2}, 0.25 for s_{T-3}, and so on. Therefore, with TD(λ) targets, the model largely trains on its own predictions rather than being grounded by the observed returns.

The extreme case is TD(0), where the target of all states, except s_{T-1}, is from the model's prediction of the next state. In the 25-6 setting, only $\sim 3\%$ of the tuples' value targets in $\text{SP}_{i=1}^{\lambda=0}$ are grounded in the return (games are about 30-40 turns), while the other $\sim 97\%$ of targets are from the model's own output. The training data distribution is then similar to the prediction distribution of the untrained model, which was visualized in the left-most histogram in Figure 4.2. Empirically, the untrained model's prediction distribution is approximately normally distributed around 0 and

slightly imbalanced. We suspect this could cause majority-class collapse, a problem we saw in Chapter 4.

5.3 TD(λ) Training Collapse Mitigation

Section 5.2.3 (TD(λ) Target Distribution) demonstrates that TD(λ) targets, especially TD(0), can lead to an imbalanced training dataset. In Chapter 4, we notice that imbalanced datasets can cause majority-class collapse during AlphaZero training. Section 4.5.1 (Majority-Class Collapse Mitigations) describes potential strategies for mitigating majority-class collapse caused by imbalanced datasets, which we can use to enable TD(λ) training. In this section, we explore two strategies to mitigate majority-class collapse in TD(λ) training.

5.3.1 Dataset Balancing with Weighted Buckets

5.3.1.1 Weighted Buckets Method

In Section 4.4.3 we prevented value head collapse by balancing the training dataset with majority-class subsampling. This is not possible with TD(λ) because the targets are not discrete. In this section, we extend our approach to continuous targets using the loss weighting technique described in Section 4.5.1.2.

We first bucket the samples (s, π, z) in the training dataset into n equal buckets based on their target values z, then pair buckets from the two ends of the distribution (bucket 1 with bucket n, bucket 2 with bucket $n-1$, etc.). To balance the dataset, we assign loss weights to each sample such that the aggregate loss contribution of both buckets in each pair is equal. An odd number of buckets will result in one unpaired bucket around 0. For example, if $n = 5$, we have buckets $[-1, -0.6)$, $[-0.6, -0.2)$, $[-0.2, 0.2)$, $[0.2, 0.6)$, and $[0.6, 1]$. If $n = 2$, we have buckets $[-1, 0)$ and $[0, 1]$.

Most class-balancing strategies balance all classes against each other because typical

classes are nominal. However, in our case, that would require extreme balancing measures (e.g. drastic subsampling or lots of synthetic data) because early in training the extreme buckets are nearly empty while most of the samples are in the middle buckets. Our ordinal buckets and approximately symmetrical distribution allow us to balance similarly sized buckets to achieve an overall balanced dataset.

Subsampling the larger bucket of each pair loses information, so instead we lower the weight of the loss of the majority bucket so that the total loss of each bucket pair is equal.

5.3.1.2 Weighted Buckets in AlphaZero with TD(λ)

In AlphaZero, the self-play tuples in SP_i^λ generated at iteration i are put into a replay buffer. To train the model at iteration i, a list of tuples is sampled from the replay buffer to create a training dataset. To balance the model's training data, we apply the weighted buckets method described above (Section 5.3.1.1) to this dataset.

To observe the effect of weighted buckets, we trained with TD(λ) targets with $\lambda \in \{0, 0.5, 0.75, 1\}$ and weighted buckets with $n \in \{0, 2, 5\}$ buckets, where 0 buckets means no balancing. We trained 50 seeds for each configuration to 15 iterations and recorded the final value head accuracy. We did not run the two weighted buckets configurations at $\lambda = 1$.

5.3.1.3 Results of Weighted Buckets in AlphaZero with TD(λ)

The results of the experiment described above (Section 5.3.1.2) show that TD(0) is unstable without balancing, while TD(0.5) and TD(1) are stable without balancing. With no balancing, TD(0) has 43 of 50 trials collapsing to a constant value head prediction of 1 or -1. With 2 buckets, 11 of 50 trials collapsed, and with 5 buckets, 9 of 50 trials collapsed. While balancing helps TD(0) training stability, it slows down learning and reduces the final accuracy after 15 iterations. These results are shown in

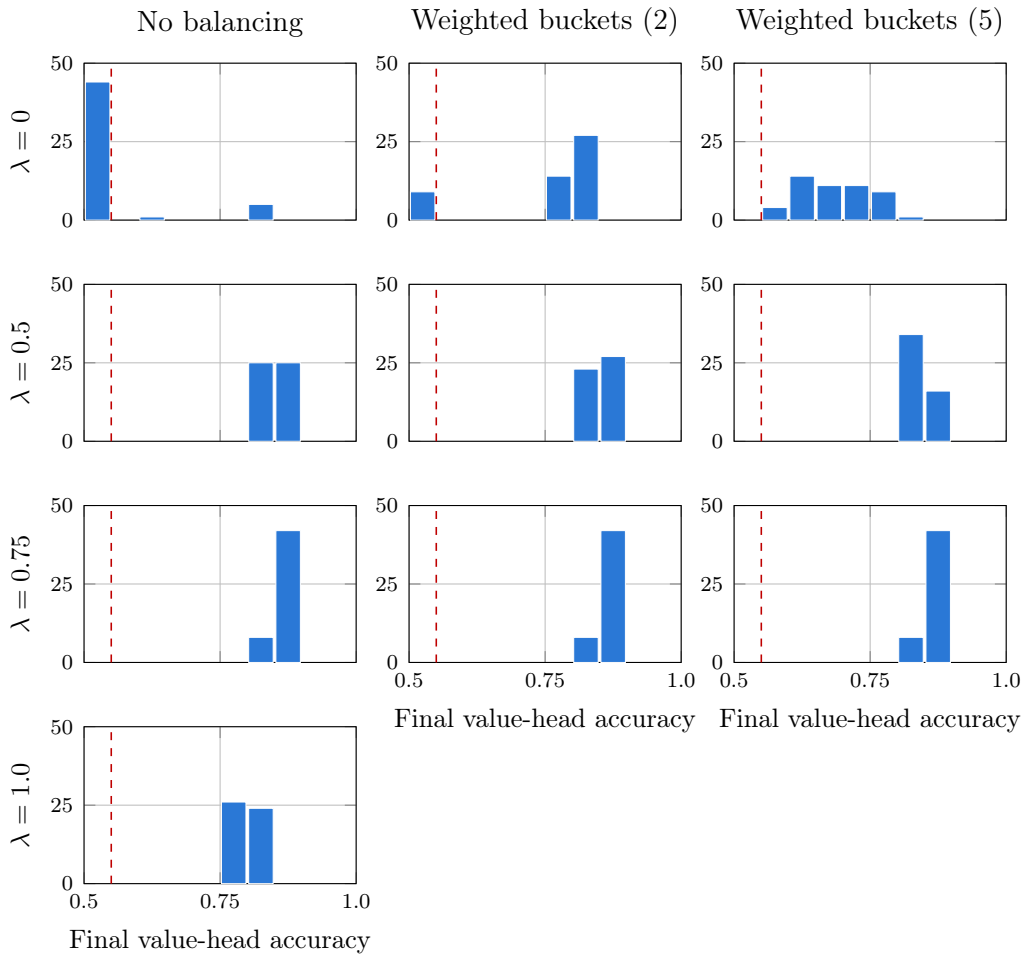


Figure 5.1: Distribution of the final value head accuracy over 50 trials for each pair of λ and dataset balancing. Columns move left to right from no balancing to weighted-bucket balancing with 2 and 5 buckets; rows move top to bottom from $\lambda = 0$ to $\lambda = 1.0$. The y axis is the number of trials in each accuracy bin; the dashed line marks the 0.55 collapse threshold. The two weighted-bucket configurations at $\lambda = 1.0$ were not run.

Figure 5.1.

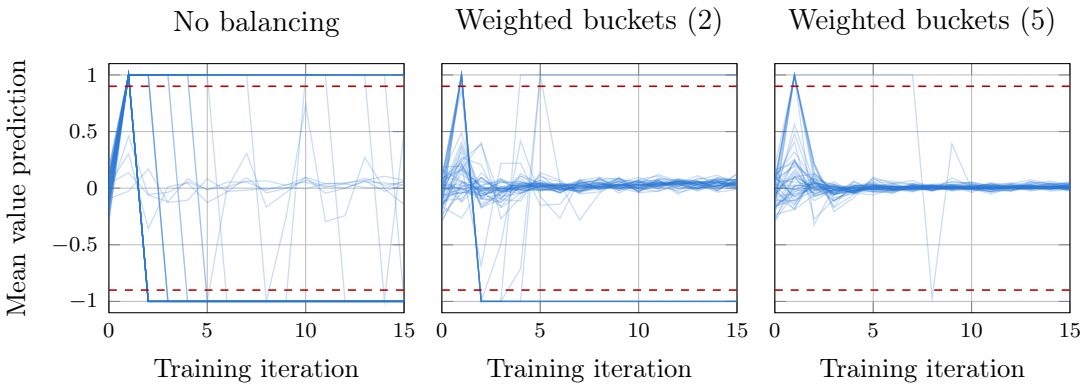


Figure 5.2: The model's mean value prediction (y-axis) over training iterations (x-axis) with TD(0) targets, one line per trial (50 trials per panel). The balance setting for each graph is indicated in their title. From left to right: no balancing, weighted-bucket balancing with 2 buckets, and with 5 buckets. The dashed lines mark a collapse threshold $|\text{mean}| = 0.9$.

We also look at the model's mean value prediction over training iterations. We notice that value head collapse happens early in training. We also notice, in some trials, the model collapses to a constant prediction of 1 or -1, only to collapse in the opposite direction, or even recover. These results are visualized in Figure 5.2.

As discussed in Section 5.2.3, the majority of TD(0)'s targets come from the model's own negated predictions. This provides a self-correcting mechanism that should rescue the model from collapse. However, the model often collapses and does not recover, despite most of its value head targets being the opposite of its current predictions. We find that tanh's pre-activation value in these unrecoverable collapsed models is in its saturated region.

The ideal weighted buckets setting seems to be 2 buckets, which retains learning ability while still preventing collapse $\sim 80\%$ of the time. Because collapses happen early in training, collapsed seeds can simply be retried with a new seed.

However, we find that weighted buckets + retries does not work to prevent collapse

on larger board sizes (36-6 and 49-4), even in the default AlphaZero setting with balanced MC targets. This motivates a more robust solution to our value head collapse problem.

5.3.2 Stabilizing the Value Head with Zero Initialization

We suggested the possibility of value head collapse being caused by training instability in Section 4.5.1.3. Following this, we tried multiple strategies to stabilize value head training, including increasing the number of residual blocks in the model, separating the value and policy head gradients with a separate backbone, increasing the turn limit to remove draws in the training dataset, gradient clipping, and zero-initialization in the value head. We found that zero-initialization solved the collapse problem in all our settings, so we adopt it as a standard feature of our network.

We zero-initialize the weights of the value head's final linear layer. This causes the model to predict 0 for all states at initialization.

We also redo the experiment from Section 4.4.2, where we train a model on the S_{train} dataset but with subsampling performed to create an imbalanced dataset. With a majority-class fraction of $p_{\text{maj}} = 0.9$ and 30 seeds, our zero-initialized value head model has 0 runs collapse and 30 runs learn successfully.

Based on our observations, we suggest that value head collapse can be caused by both imbalanced datasets and training instability in the neural network. While balancing the dataset can help, a properly designed neural network and training procedure can prevent collapse even on extremely imbalanced datasets. In extreme cases, training instability causes model collapse even on balanced datasets. This is likely the case in our larger board setting, possibly because the number of parameters in the value head scales quadratically with board size. We recommend treating value head collapse as a neural network design problem first, then a dataset or loss problem.

5.4 Method

To train AlphaZero with TD(λ) targets, we use the training and evaluation procedure described in Section 5.2.2 (TD(λ) Targets in AlphaZero). We additionally use zero-initialization in the value head to prevent collapse.

In addition to standard metrics like accuracy and win rate, we also report a converged metric, defined as the average value over the final 20% of training iterations. We use this to compare the final performance of models trained with different λ values.

We train AlphaZero with TD(λ) targets on Chinese Checkers on board sizes 16-3, 25-6, 36-6, 49-4 with λ values 0.0, 0.5, 0.75, 0.9, 1.0. We then look at the value head accuracy, the player win rate, and training data accuracy. The accuracy metrics are not available in the 36-6 setting because the solve data was too large for our scripts. The number of seeds trained varies and is reported in the figures. All other hyperparameters are the same as in the default AlphaZero setting. Figures for additional metrics (policy head accuracy and policy target accuracy) are included in Appendix C.

5.5 Results

5.5.1 Value Head Accuracy

In this section, we analyze the value head accuracy of the trained models over training iterations and the final converged value head accuracy.

Value head accuracy converges faster with higher λ values, in all four settings. However, there is a drop in accuracy for $\lambda = 1.0$ (the default AlphaZero λ value) in the 25-6 and 49-4 settings. This is most evident in the 25-6 setting, where $\lambda = 1.0$ converges to $\sim 86\%$, while the other λ values converge to $\sim 90\%$. This also occurs in the 49-4 setting on just the seen states, where $\lambda = 1.0$ converges to $\sim 91\%$, while the other λ values converge to $\sim 93.5\%$. In the 16-3 setting, all λ values converge to a

similar accuracy. The λ values that converge to the highest final accuracy are in the range of 0.5 to 0.9, depending on the setting.

Board	States	$\lambda < 1$	$\lambda = 1$	TD(1) difference
16-3	Seen	92.6–93.1	92.0	-0.7 to -1.1
	Random	91.6–92.2	91.5	-0.1 to -0.7
25-6	Seen	89.9–90.7	85.6	-4.3 to -5.1
	Random	89.5–90.3	85.5	-4.0 to -4.8
49-4	Seen	92.9–94.2	91.3	-1.7 to -3.0
	Random	87.2–88.3	87.2	0.0 to -1.1

Table 5.1: Converged value head accuracy (%), averaged over the last 20% of training iterations. The $\lambda < 1$ column gives the converged accuracy range over $\lambda \in \{0, 0.5, 0.75, 0.9\}$. The $\lambda = 1$ column gives the converged accuracy for $\lambda = 1$. The last column subtracts each end of the range from $\lambda = 1$, so a negative difference means TD(1) is worse. Averaged over 25 seeds for 16-3 and 25-6, and 12 seeds for 49-4.

To visualize these results, we take the value head accuracy curves from each λ setting and plot them on the same graph to compare the effect of λ on value head accuracy. We also take the average accuracy of each λ's converged models (the last 20% iterations) and plot them below the accuracy curves to visualize the effect of λ on final value head accuracy. For each λ setting, we plot the value head accuracy on both the seen states and random states sets. These graphs are shown in Figures 5.3 to 5.5. The difference in converged accuracy between λ's < 1 and $\lambda = 1.0$ for all board sizes (16-3, 25-6, and 49-4) and state sets (seen and random) is shown in Table 5.1.

5.5.2 Player Win Rate

In this section, we analyze the player win rates of the trained players over training iterations and the final converged player win rate.

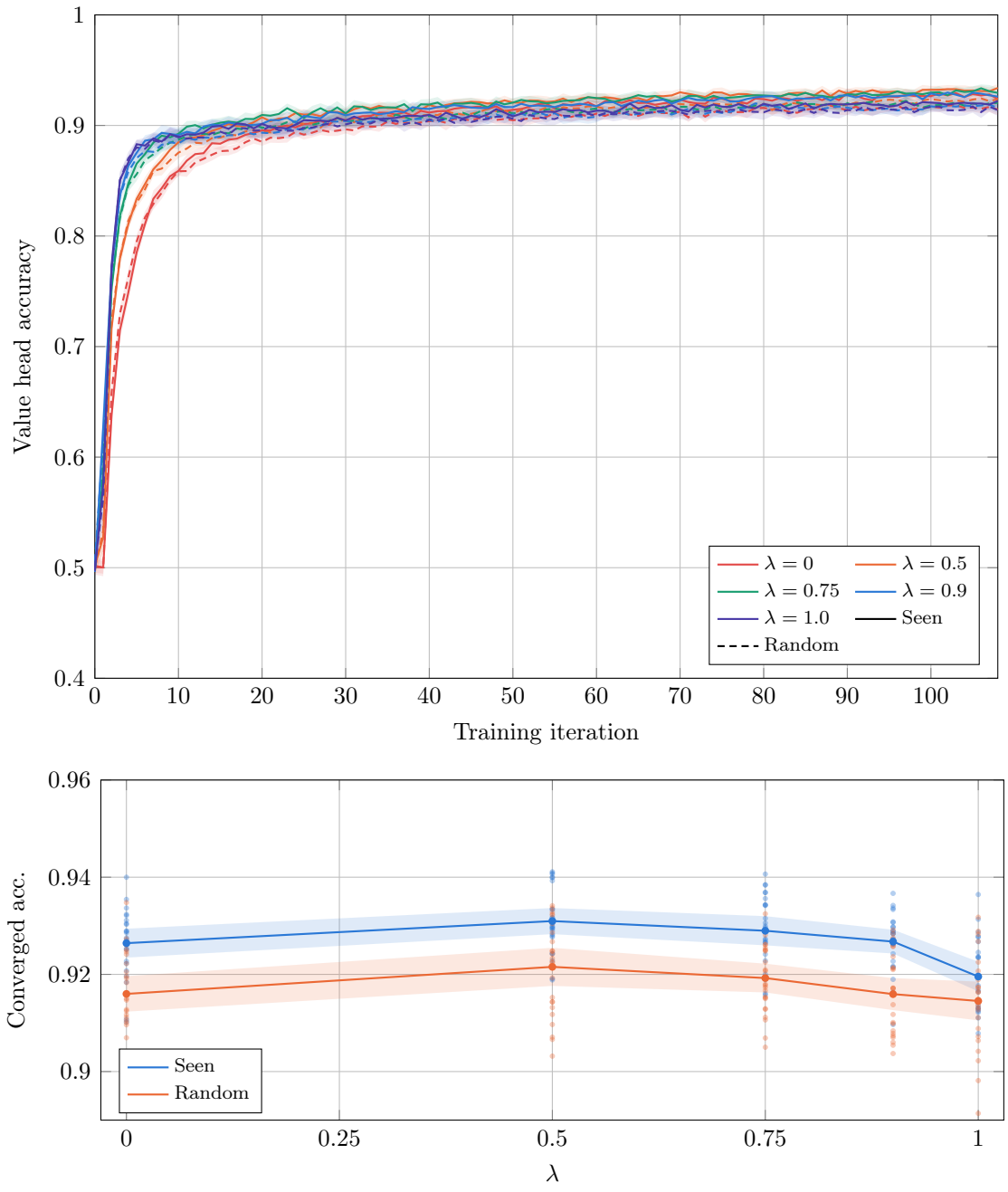


Figure 5.3: Value head accuracy over training on 16-3 across λ values, on seen (solid) and random (dashed) states with shaded 95% CI bands. Bottom: the converged (last-20%) accuracy over λ, with 95% CI bands and the per-trial converged values as dots. Averaged over 25 seeds.

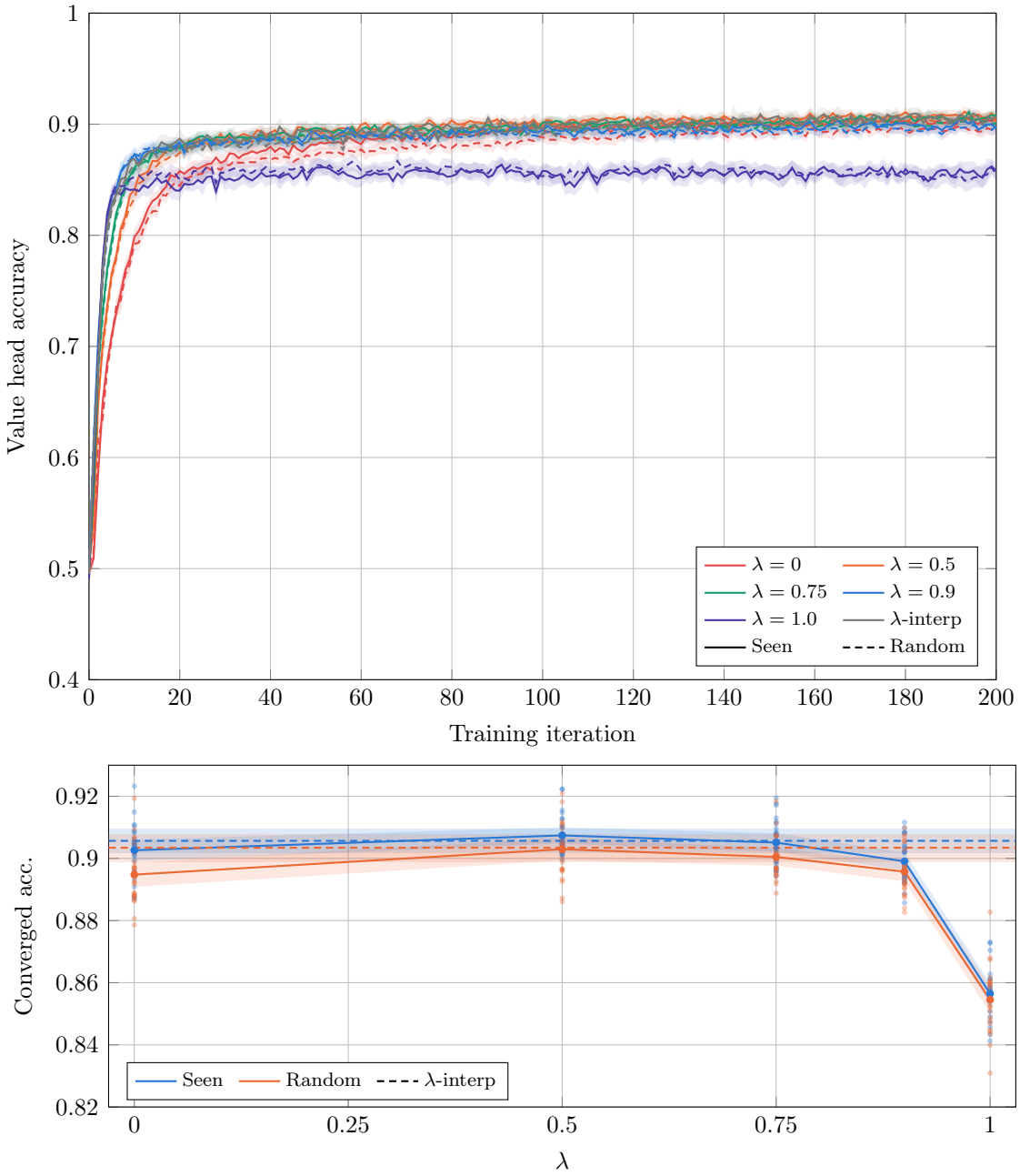


Figure 5.4: Value head accuracy over training on 25-6 across λ values (plus the lambda-interpolation setting), on seen (solid) and random (dashed) states with shaded 95% CI bands. Bottom: the converged (last-20%) accuracy over λ, with 95% CI bands and the per-trial converged values as dots; dashed lines mark the lambda-interpolation setting. Averaged over 25 seeds.

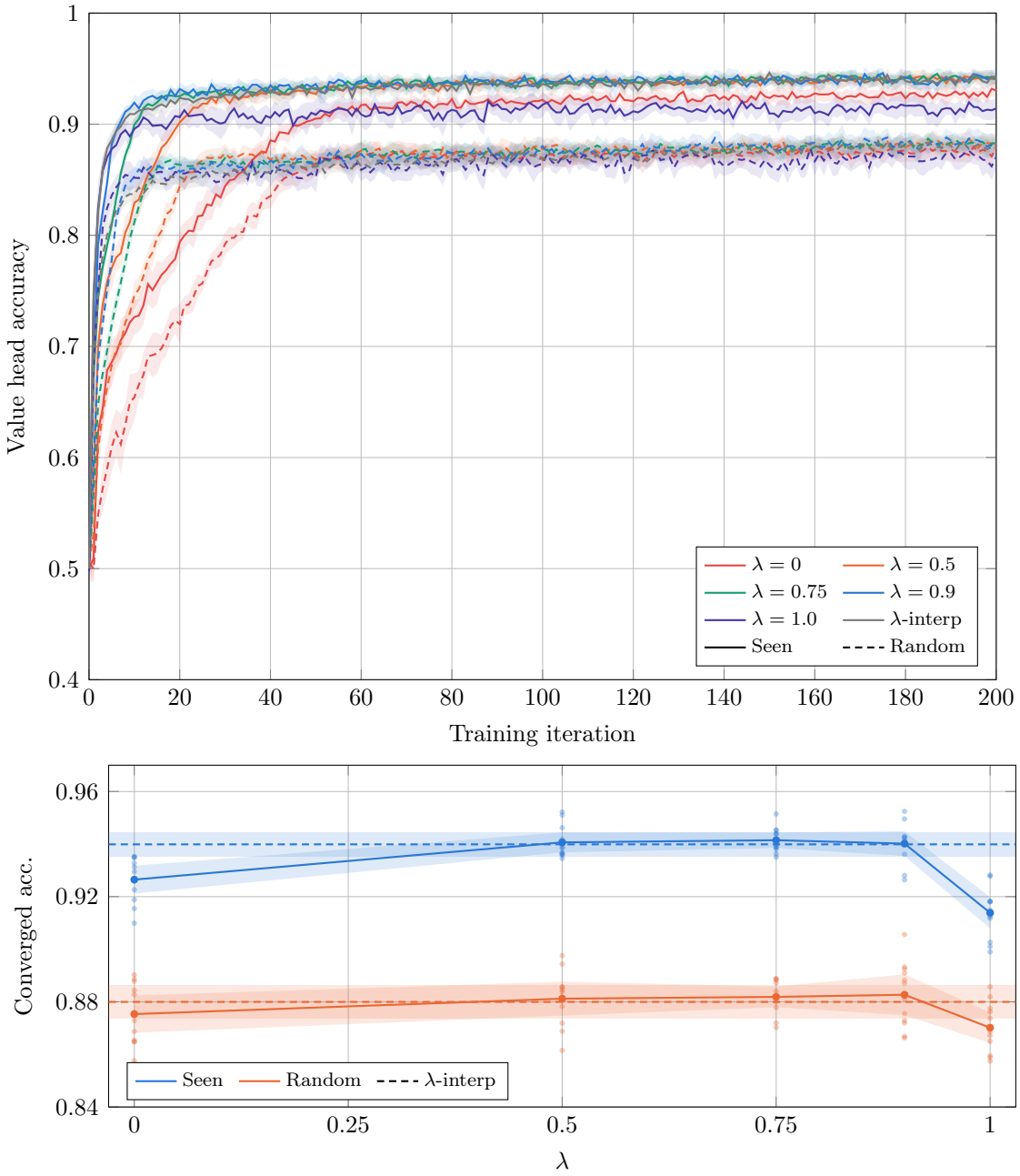


Figure 5.5: Value head accuracy over training on 49-4 across λ values (plus the lambda-interpolation setting), on seen (solid) and random (dashed) states with shaded 95% CI bands. Bottom: the converged (last-20%) accuracy over λ, with 95% CI bands and the per-trial converged values as dots; dashed lines mark the lambda-interpolation setting. Averaged over 12 seeds.

The results are similar to the value head accuracy results. Higher λ values converge faster. However, we see a drop in converged win rate for $\lambda = 1.0$ in the 25-6, 36-6, and 49-4 settings. In 25-6 for example, the λ values 0.0, 0.5, 0.75, and 0.9 all converge to a similar P1 win rate of $\sim 99\%$, while the $\lambda = 1.0$ P1 win rate converges to $\sim 92\%$, a 7% drop. Similarly, the P2 win rate for $\lambda = 1.0$ is lower than the other λ values by $\sim 10\%$ ($\sim 95\%$ to $\sim 85\%$). In the 16-3 setting, all λ values converge to a similar P1 win rate of $\sim 99\%$ and P2 win rate between 81% and 86%.

Board	Player	$\lambda < 1$	$\lambda = 1$	TD(1) difference
16-3	P1	99.0–99.4	98.9	-0.1 to -0.5
	P2	81.3–86.4	83.2	$+1.9$ to -3.2
25-6	P1	98.4–99.0	92.3	-6.1 to -6.7
	P2	94.5–95.5	83.9	-10.5 to -11.5
36-6	P1	94.4–98.2	81.9	-12.5 to -16.4
	P2	87.4–91.2	58.0	-29.3 to -33.2
49-4	P1	99.5–99.9	96.5	-3.0 to -3.3
	P2	81.9–87.0	68.0	-13.8 to -19.0

Table 5.2: Converged player win rate (%) against the baseline player, averaged over the last 20% of training iterations. The $\lambda < 1$ column gives the converged win rate range over $\lambda \in \{0, 0.5, 0.75, 0.9\}$. The $\lambda = 1$ column gives the converged win rate for $\lambda = 1$. The last column subtracts each end of the range from $\lambda = 1$, so a negative difference means TD(1) is worse. Averaged over 15 seeds for 16-3 and 25-6, and 12 seeds for 36-6 and 49-4.

To visualize these results, we repeat the same procedure as in the previous section, but with player win rate instead of value head accuracy. We also plot the P1 and P2 win rates instead of the seen and random accuracies. These graphs are shown in Figures 5.6 to 5.9. The difference in converged win rate between $\lambda < 1$ and $\lambda = 1.0$ for all board settings (16-3, 25-6, 36-6, and 49-4) and both P1 and P2 is shown in

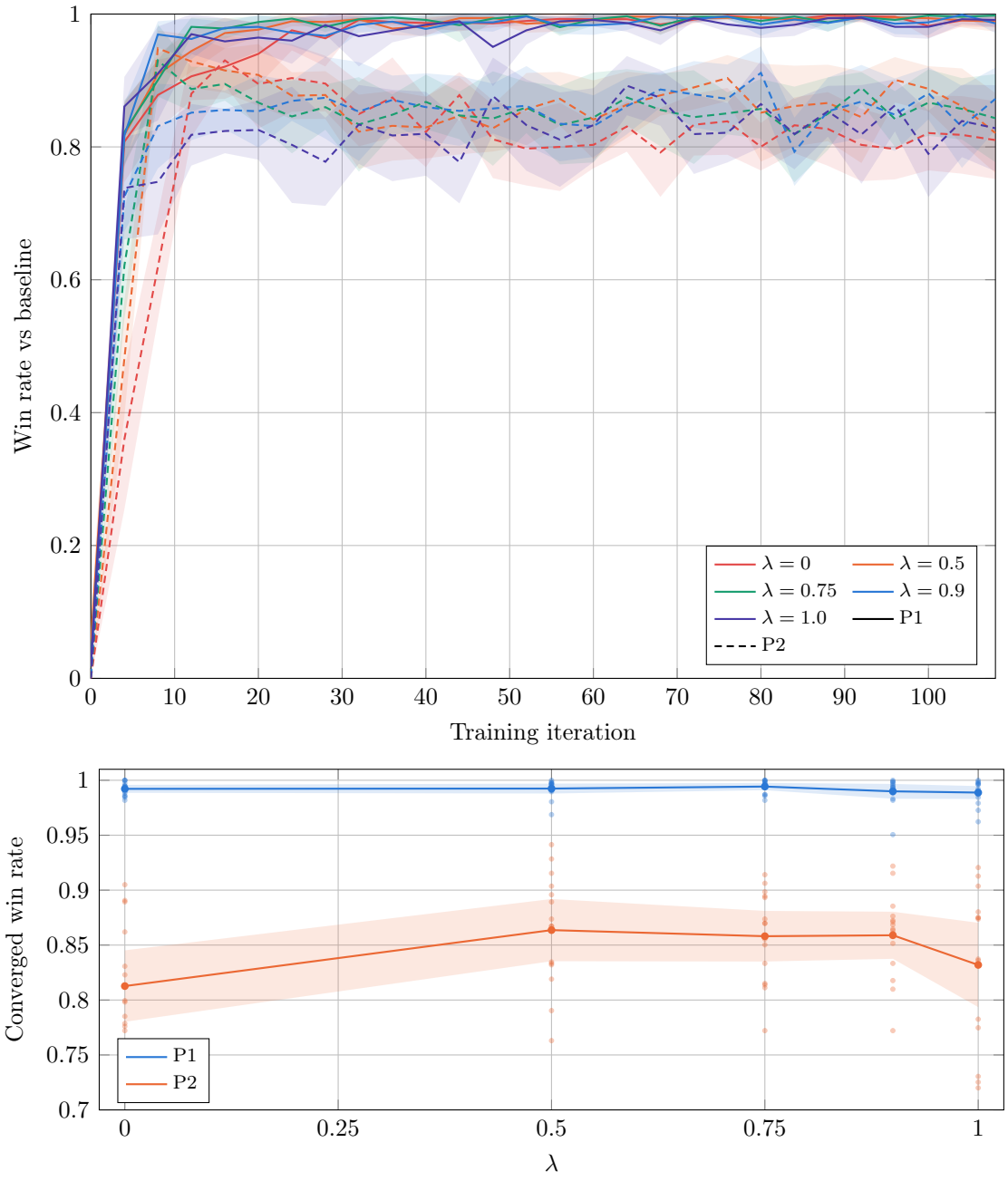


Figure 5.6: Player win rate against the baseline over training on 16-3, across λ values, as first (P1, solid) and second (P2, dashed) player, with shaded 95% CI bands. Bottom: the converged (last-20%) win rate over λ, with 95% CI bands and the per-trial converged values as dots. Averaged over 15 seeds.

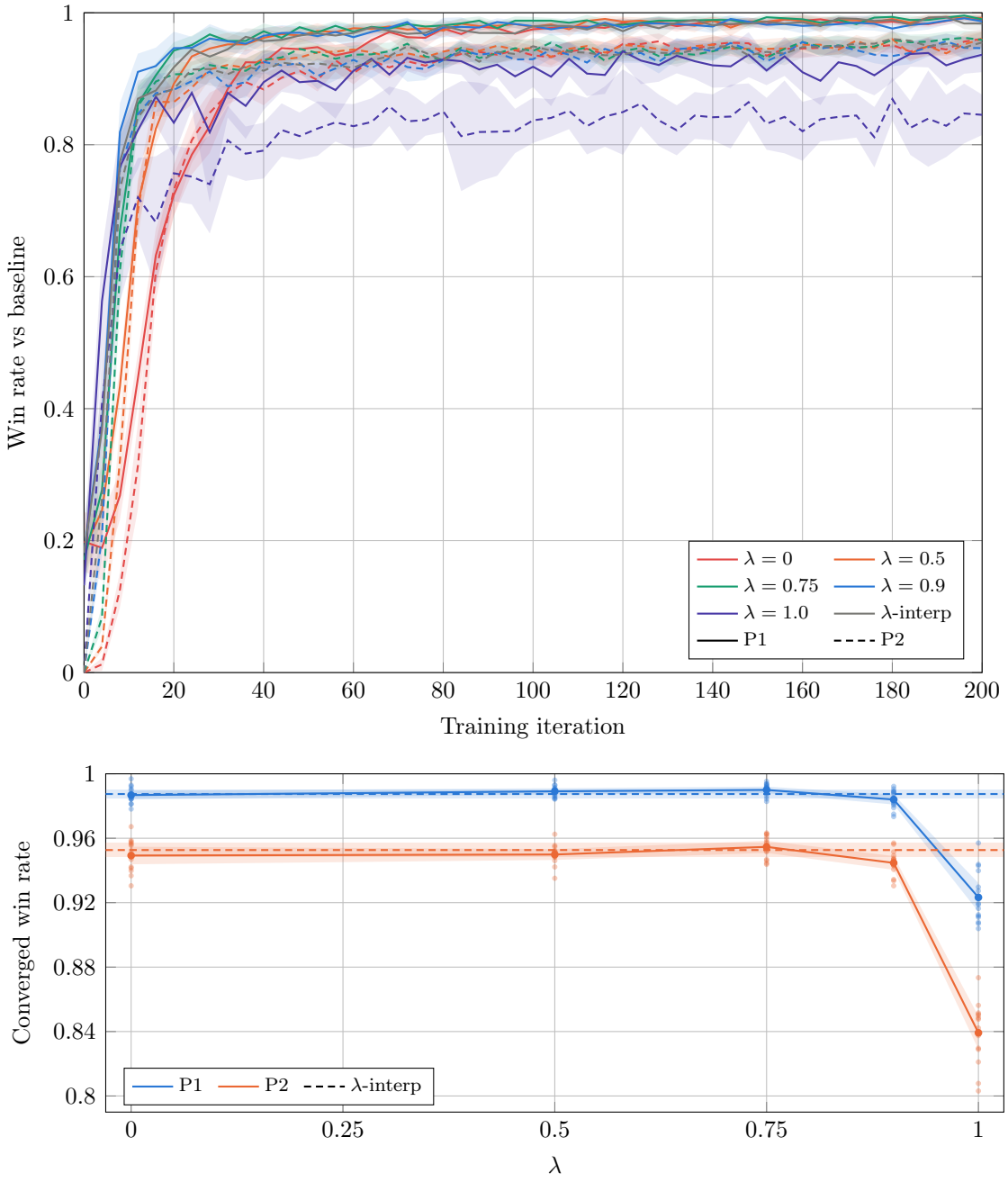


Figure 5.7: Player win rate against the baseline over training on 25-6 across λ values (plus the lambda-interpolation setting), as first (P1, solid) and second (P2, dashed) player, with shaded 95% CI bands. Bottom: the converged (last-20%) win rate over λ, with 95% CI bands and the per-trial converged values as dots; dashed lines mark the lambda-interpolation setting. Averaged over 15 seeds.

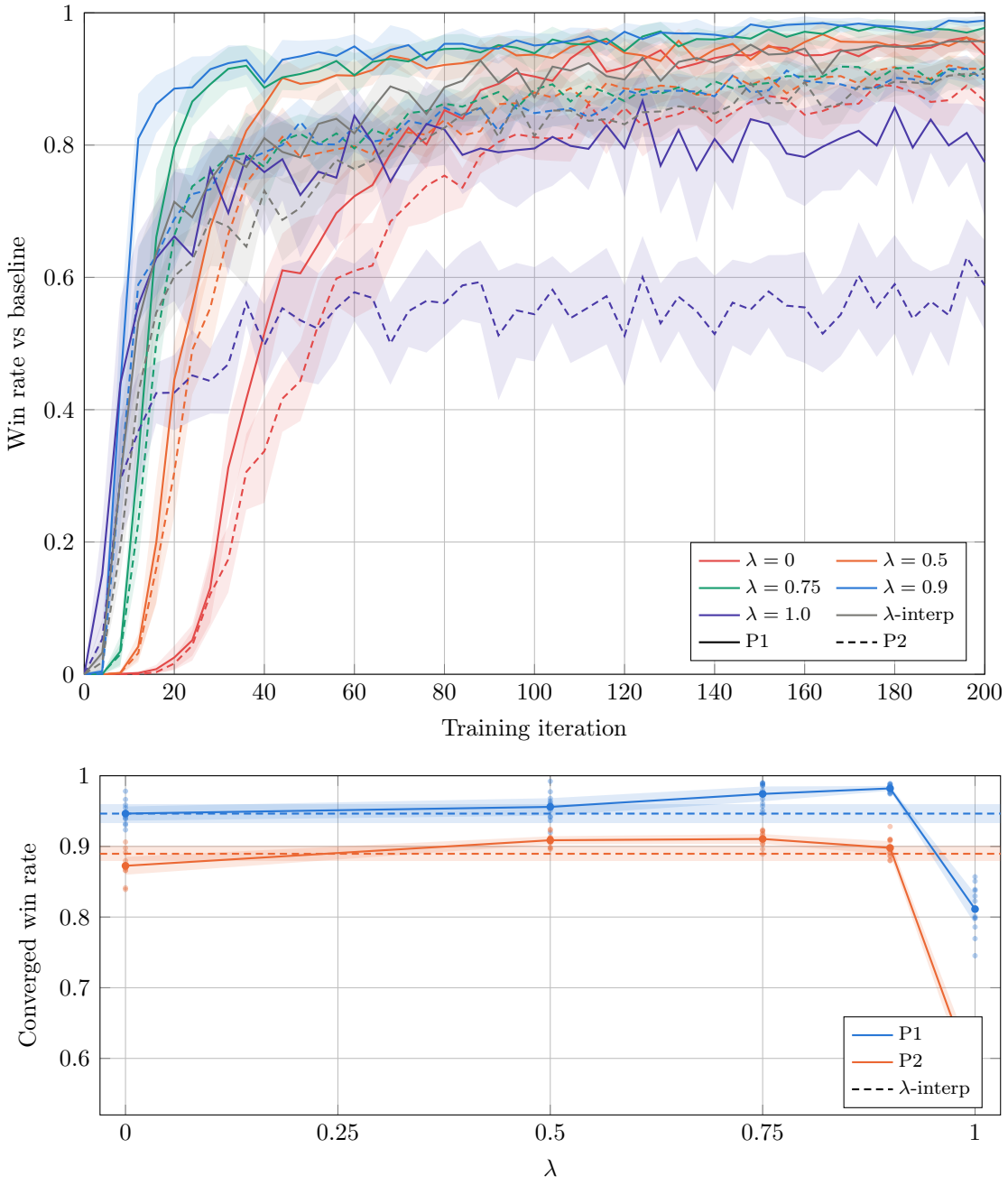


Figure 5.8: Player win rate against the baseline over training on 36-6 across λ values (plus the lambda-interpolation setting), as first (P1, solid) and second (P2, dashed) player, with shaded 95% CI bands. Bottom: the converged (last-20%) win rate over λ, with 95% CI bands and the per-trial converged values as dots; dashed lines mark the lambda-interpolation setting. Averaged over 12 seeds.

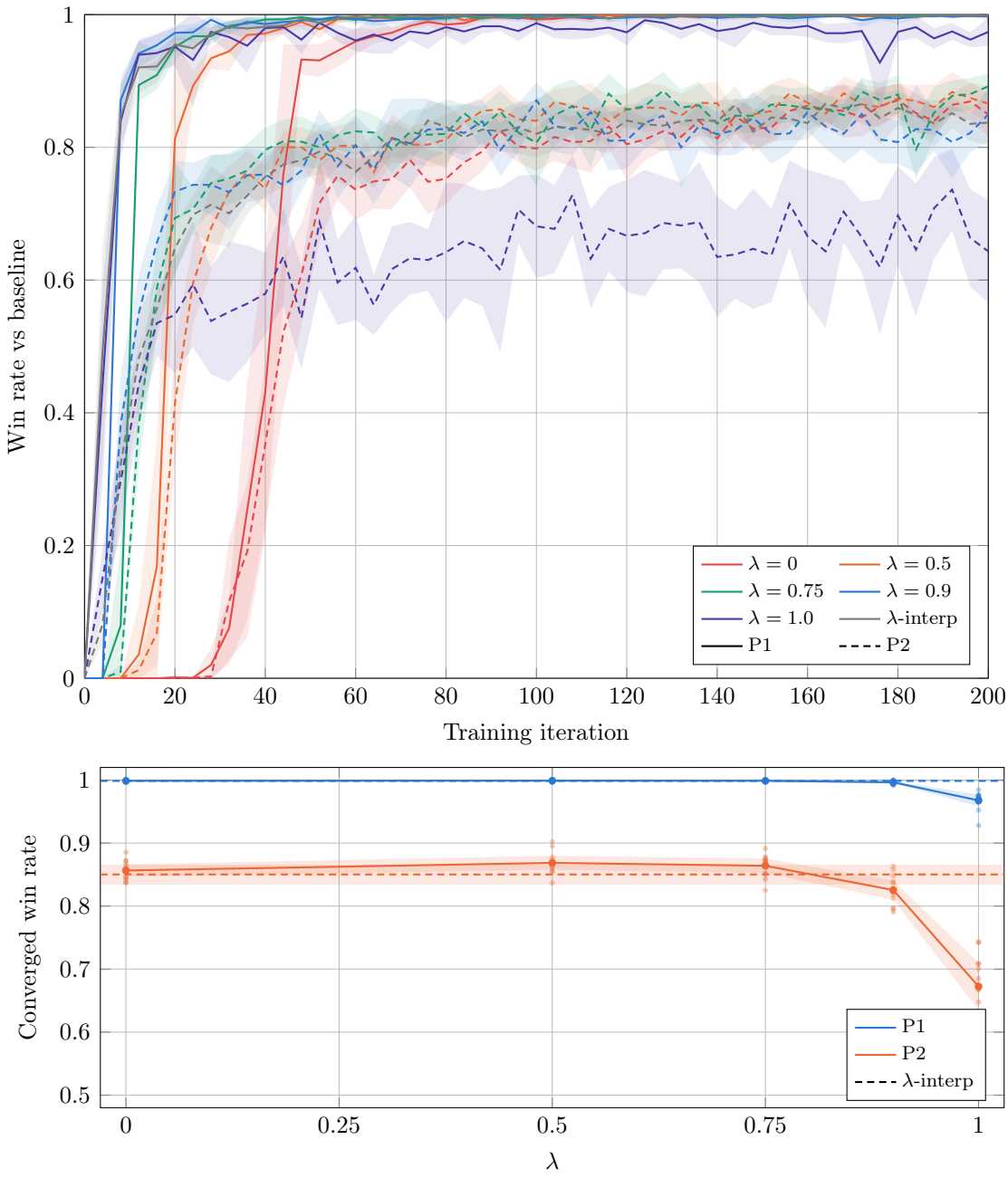


Figure 5.9: Player win rate against the baseline over training on 49-4 across λ values (plus the lambda-interpolation setting), as first (P1, solid) and second (P2, dashed) player, with shaded 95% CI bands. Bottom: the converged (last-20%) win rate over λ, with 95% CI bands and the per-trial converged values as dots; dashed lines mark the lambda-interpolation setting. Averaged over 12 seeds.

Table 5.2.

5.5.3 Value Target Accuracy

In this section, we analyze the accuracy of the value head's training data, the accuracy of the value targets, over training iterations.

For all board sizes, the value target accuracy curves show that higher λ values converge faster, but to a lower final accuracy, while lower λ values converge slower, but to a higher final accuracy. For example, in the 25-6 setting, the converged value target accuracy for λ values 0.0, 0.5, 0.75, 0.9, and 1.0 are $\sim 85.7\%$, $\sim 84.7\%$, $\sim 80.7\%$, $\sim 74.7\%$, and $\sim 68.5\%$ respectively.

To visualize these results, we repeat the same procedure as in the previous section, but with value target accuracy instead of player win rate. The training data accuracy curves are shown in Figures 5.10 to 5.12.

5.6 Extensions to TD(λ) in AlphaZero

Upon seeing the above results, we created two modifications to improve TD(λ) training in AlphaZero: TD(λ) interpolation and TD(λ) with refreshes.

5.6.1 TD(λ) Interpolation

Motivation The training data accuracy curves suggest a principled way to choose λ – pick the λ value that would produce the highest training data accuracy. Figures 5.10 to 5.12 show that this ideal λ value changes over the course of training. For example, in 49-4 the ideal λ value is $\lambda = 1.0$ until iteration 7, then it is $\lambda = 0.9$ until iteration 13, $\lambda = 0.75$ until iteration 26, $\lambda = 0.5$ until iteration 57, and finally $\lambda = 0.0$ until the end of training (though 0.5 and 0.0 overlap significantly). This suggests that we can improve TD(λ) training by starting with a high λ value and decreasing it over time.

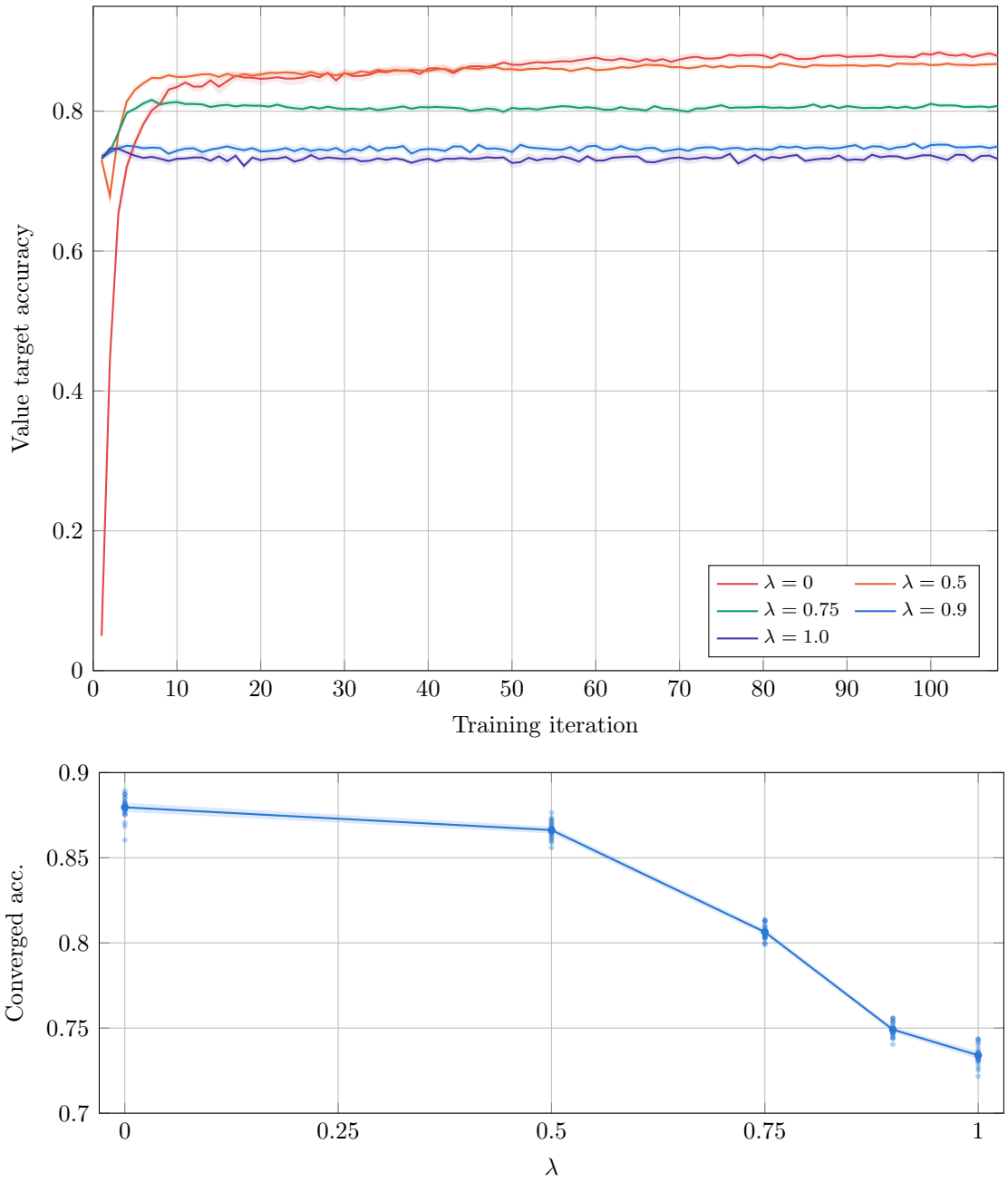


Figure 5.10: Accuracy of the value targets over training on 16-3, one line per λ with shaded 95% CI bands. These are the targets the model trains on (the TD(λ) targets; at $\lambda = 1$ they are the MC game outcomes). Bottom: the converged (last-20%) accuracy over λ, with 95% CI bands and the per-trial converged values as dots. Averaged over 25 seeds.

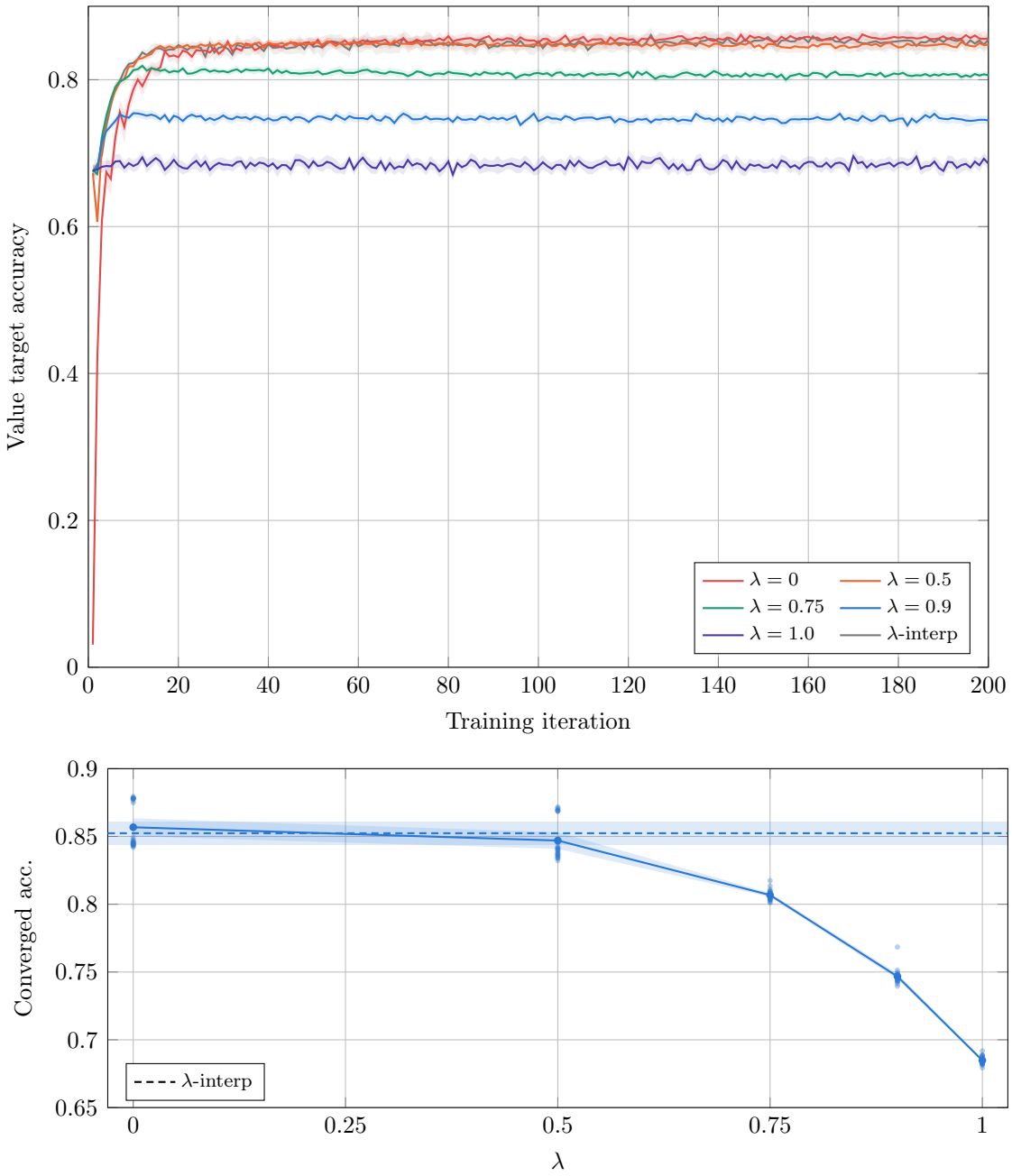


Figure 5.11: Accuracy of the value targets over training on 25-6, one line per λ (plus the lambda-interpolation setting) with shaded 95% CI bands. These are the targets the model trains on (the TD(λ) targets; at $\lambda = 1$ they are the MC game outcomes). Bottom: the converged (last-20%) accuracy over λ, with 95% CI bands and the per-trial converged values as dots. Averaged over 25 seeds.

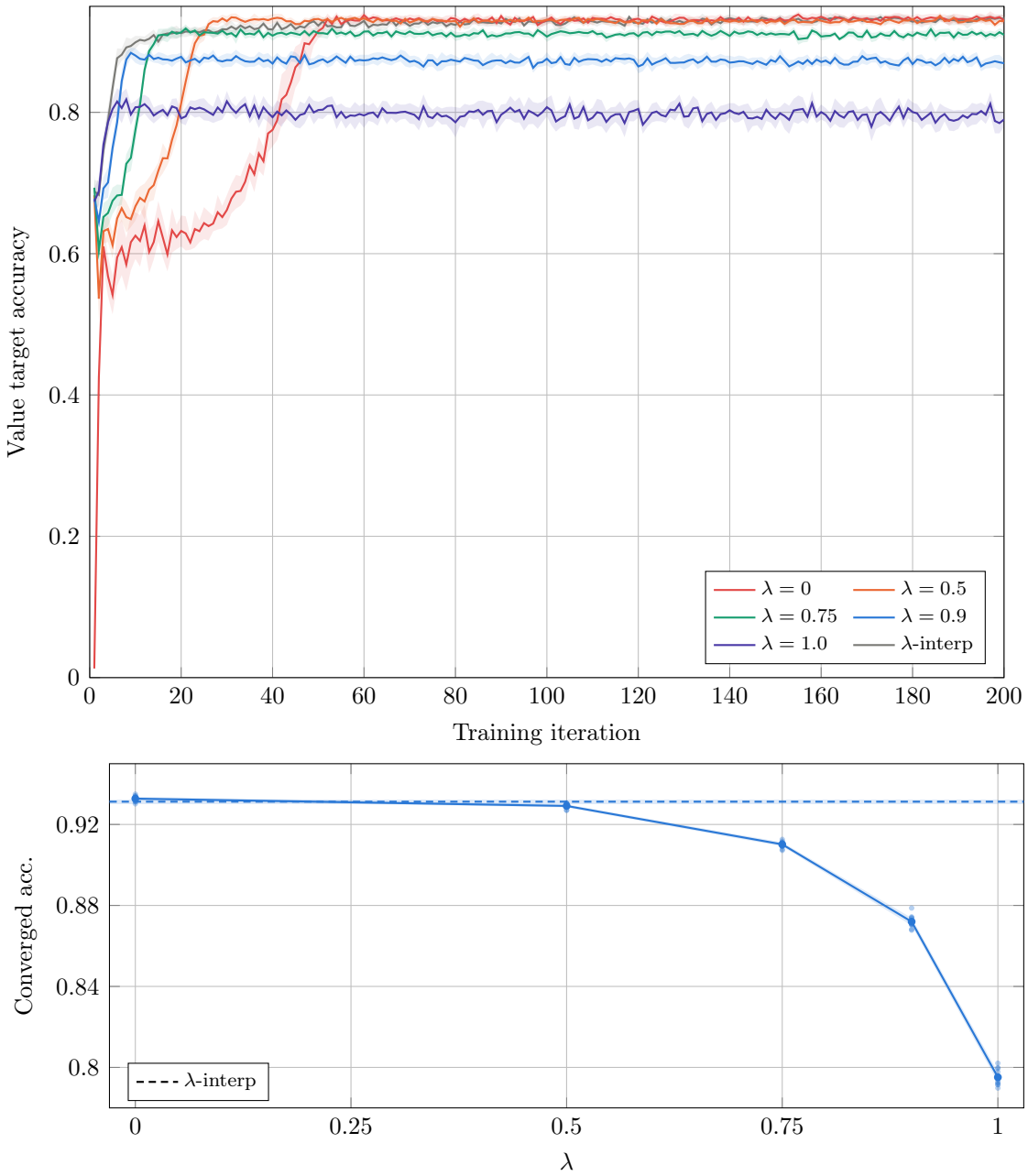


Figure 5.12: Accuracy of the value targets over training on 49-4, one line per λ (plus the lambda-interpolation setting) with shaded 95% CI bands. These are the targets the model trains on (the TD(λ) targets; at $\lambda = 1$ they are the MC game outcomes). Bottom: the converged (last-20%) accuracy over λ, with 95% CI bands and the per-trial converged values as dots; the dashed line marks the lambda-interpolation setting. Averaged over 12 seeds.

Method This motivates our first modification, TD(λ) interpolation. In TD(λ) interpolation, we linearly interpolate λ from 1 to 0 over the course of training. This allows us to start with a high λ value to improve convergence speed, and end with a low λ value to improve final accuracy.

We run TD(λ) interpolation on board sizes 25-6, 36-6, and 49-4. For all these board sizes, we interpolate from 1 to 0 in 25 iterations.

Results TD(λ) interpolation has the high convergence speed of $\lambda = 1.0$ and high final value head accuracy, value target accuracy, and win rate of $\lambda = 0.0$. These results can be seen in Figures 5.4, 5.5, 5.7 to 5.9, 5.11 and 5.12.

5.6.2 TD(λ) with Refreshes

Motivation Our results show that TD(0) converges slower than TD(1), and that this slow convergence worsens with board size. One hypothesis for this is slower credit propagation. In TD(1), the value target for all states s_t in a game is G_t, the game outcome. In TD(0), $\hat{v}_\theta$ only sees the game outcome in the target for s_{T-1}. The value target for the rest of the states $s_0, \ldots, s_{T-2}$ is from the model's own estimates. Therefore, in iteration 1, $\hat{v}_\theta$ only learns to estimate the value on states at time $T-1$. The target for the rest of the states are from the model's initialization noise. It is only in iteration 2, when the model has an estimate for some states at $T-1$, that it can learn about states at $T-2$. Then in iteration 3, it can learn about states at $T-3$, and so on. Only one credit propagation step is performed per iteration.

Method This hypothesis motivates our second modification to AlphaZero with TD(λ), TD(λ) with refreshes. This setting performs multiple credit propagation steps per iteration by recalculating the TD(λ) targets multiple times per iteration. In iteration 1, the model learns to estimate the value of states at $T-1$. Then the targets

are recalculated using this newly trained model, so that it can learn to estimate the value of states at $T-2$ using these refreshed targets.

In the default AlphaZero with TD(λ) setting, we calculate the targets G_t^λ for all states in SP_i^λ at iteration i using the model from the previous iteration, $\hat{v}_{\theta_{i-1}}$. These targets are then put into the replay buffer and the model is trained on a dataset sampled from the replay buffer for one epoch. This is then repeated for each iteration. In TD(λ) with refreshes, the model is trained on the dataset for one epoch, then G_t^λ is recalculated for all states in the replay buffer using this newly trained model, and the model is trained on these new targets for another epoch. This is repeated for a total of k epochs per iteration. We call this TD(λ) with refreshes because the targets are refreshed after each epoch of training.

We run TD(λ) with refreshes on the board size 36-6 with $\lambda = 0.0$ and $k \in \{1, 2, 4, 8\}$. We choose board size 36-6 because of its high difference in convergence rate between TD(0) and TD(1). We choose $\lambda = 0.0$ because its convergence is the slowest and its simplicity makes it easier to analyze. We choose $k \in \{1, 2, 4, 8\}$ to test the effect of increasing the number of epochs per iteration.

Results TD(λ) refreshes improves convergence speed for $\lambda = 0.0$ at the beginning of training. With $k = 8$, TD(0) with refreshes converges at a similar speed as TD(1), and converges to a win rate similar to TD(0). Both $k = 4$ and $k = 2$ also improve convergence speed, but not as much as $k = 8$, and converge to a win rate similar to TD(1) or worse. TD(λ) with refreshes also introduces instability. This instability results in a larger spread in the converged win rate of the trained models.

These results can be seen in Figure 5.13, which shows the win rate over training and the converged win rate of each setting. We also include regular TD(0) and TD(1) for comparison, and a control setting of TD(0) with no refreshes and 8 epochs of training

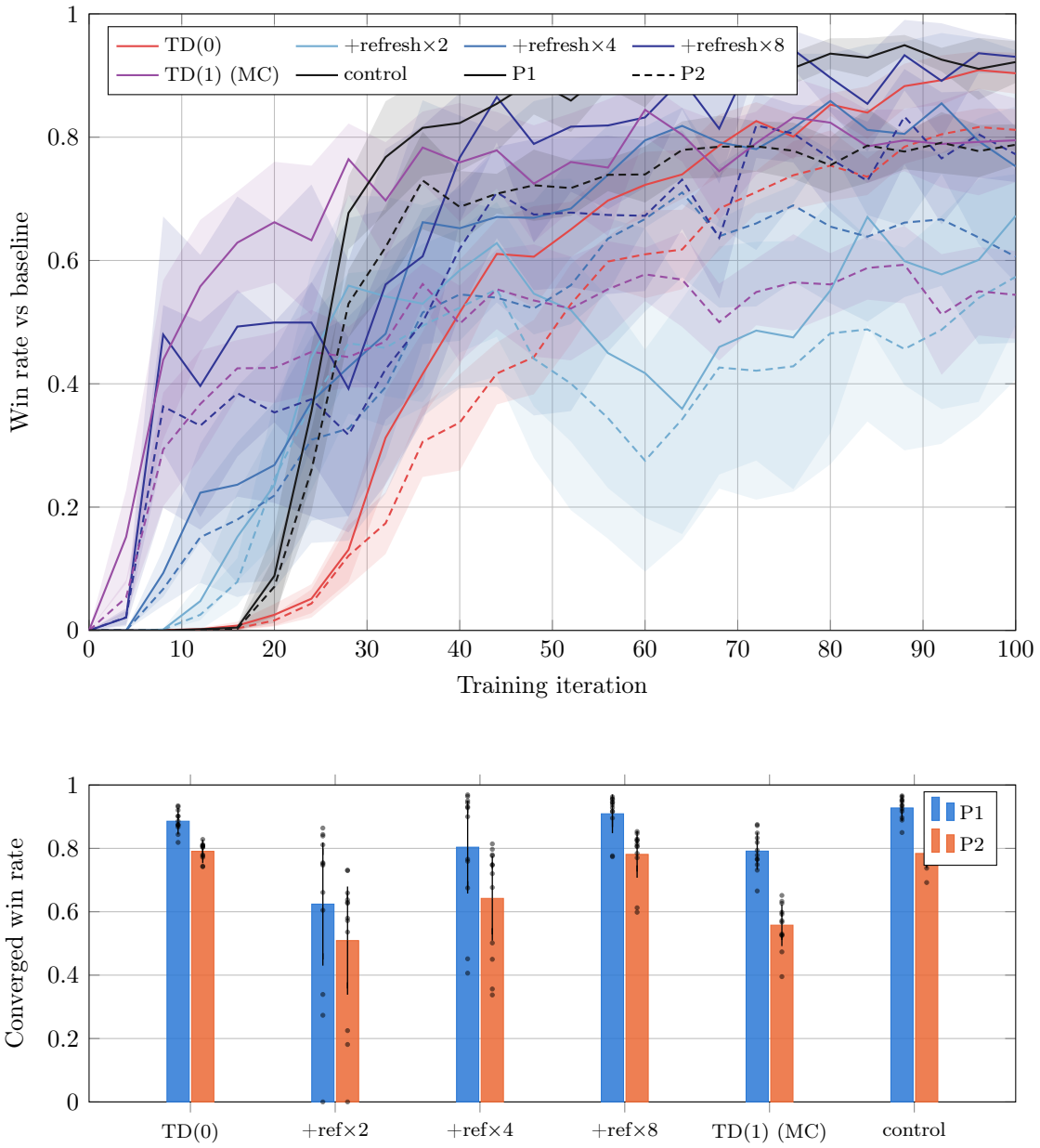


Figure 5.13: TD(λ) with refreshes win rate against the baseline over training on 36-6 across different values of k, as first (P1, solid) and second (P2, dashed) player, with shaded 95% CI bands. Additionally, TD(0), TD(1), and TD(0) with 8 non-refresh epochs are included for comparison. Bottom: the converged (last-20%) win rate as a grouped bar chart with 95% error bars, over each setting, with per-trial converged values as dots. Averaged over 12 trials for all six settings.

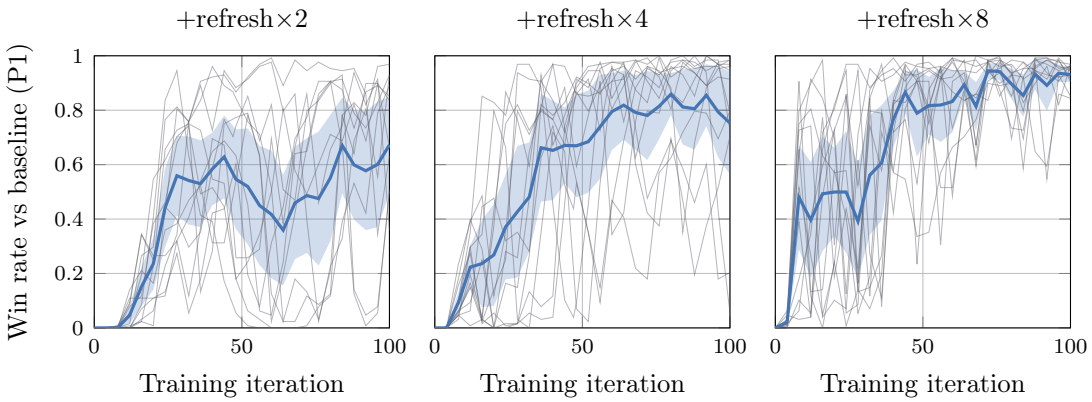


Figure 5.14: The three TD(λ) with refreshes settings on 36-6, $k \in \{2, 4, 8\}$. Each faint line is the P1 win rate over training of one trial; the thick line is the mean over trials with a shaded 95% CI band. Averaged over 12 seeds.

in each iteration. The instability can be seen in Figure 5.14, which shows each trial's win rate over training.

5.7 Discussion & Related Work

The above results show that our AlphaZero with TD(λ) method improves the final value head accuracy and win rate of the trained models. In one case, the win rate improved by 30% (36-6 P2 win rate). While the tradeoff for using TD(λ) is a slower convergence speed, TD(λ) interpolation effectively mitigates this tradeoff. TD(λ) interpolation achieves both the high convergence speed of TD(1) and the high final accuracy of TD(λ).

5.7.1 Bias and Variance in TD(λ)

MC/TD(1) targets are unbiased, but have high variance. TD(0) targets are biased, but have low variance [3]. Conventional wisdom suggests that there is a tradeoff between bias and variance, where low-bias, high-variance targets need more samples to average out noise but converge to a better final estimate, while high-bias, low-variance targets converge quickly but plateau due to the bias in the targets [63]. Our results show the

opposite. TD(0) converges slower than TD(1), yet to a better final model. In this section, we explore one hypothesis for this phenomenon.

We hypothesize that game length is the key factor in this phenomenon, as we discuss below.

5.7.1.1 Convergence Speed & Credit Propagation

We hypothesize a mechanism for TD(0)'s slow convergence in this section. TD(0)'s slow convergence is likely due to slow credit propagation, a hypothesis we described in Section 5.6.2 (TD(λ) with Refreshes). TD(0) is only able to learn a value estimate for states at $T-1$ in iteration 1, then states at $T-2$ in iteration 2, and so on. This is because, to estimate the value of a state at time t, TD(0)'s $\hat{v}_\theta$ needs the ability to estimate the value of states at time $t+1$. Each iteration only performs one credit propagation step, so it takes T iterations to learn to estimate the value of states at time 0. In contrast, TD(1) is able to learn a value estimate for all states in a game in one iteration because the game outcome is propagated to the target of all states in the game.

Game length increases with board size. Our results show that the difference in convergence speed between TD(0) and TD(1) also increases with board size. Credit propagation suggests a potential mechanism – the longer the game, the more credit propagation steps are required to learn a good value estimate in earlier states, and therefore TD(0) converges slower.

Our TD(λ) with refreshes method provides some empirical evidence that slow convergence is due to slow credit propagation. By performing multiple credit propagation steps per iteration, TD(λ) with refreshes improves convergence speed.

The above analysis ignores some factors, like similar states and function approximation, or states that appear at multiple time steps within and across games. This is a limitation of our analysis, but the effect of these factors may be small compared to

the effect of credit propagation.

5.7.1.2 Final Model Quality & Variance over Game Length

We hypothesize a mechanism for that TD(1) converging to a lower final model quality compared to TD(0) in this section. This is likely due to high variance in the targets. TD(1) targets are the game outcome, which is a single sample of the return distribution. While the targets for states at time $T-1$ are low variance, variance increases as distance from T increases. The variance at a time t is determined by how many possible games the behaviour policy can generate. This suggests that the variance increases superlinearly with distance from T, and games with longer game lengths have higher variance. While an approximate value estimate can be learned for all states quickly, approaching the true value estimate takes many more samples – this number increasing superlinearly with game length.

In our results, we see that TD(1)'s final model quality does not increase with the number of iterations. This suggests another limiting factor – behaviour policy churn. $\hat{v}_\theta$ does not get enough samples to learn an accurate value estimate at iteration i. Then the behaviour policy changes at iteration $i+1$, and $\hat{v}_\theta$ is unable to converge again. For TD(1), the number of samples within iterations may be more important than the number of samples over training to learn an accurate value estimate.

One limitation to this analysis is that we have not run experiments to support our hypothesis. We leave this as future work.

5.7.2 Future Work

Strengthening Our Results One limitation of our results is the narrow scope of our experiments. We run AlphaZero experiments on a single game – Chinese Checkers – and a small set of hyperparameters. We can strengthen our results with additional experiments. We can run more seeds and train for more iterations, though our results

suggest we have hit diminishing returns here. More importantly, we can attempt to generalize these results to other settings. We can test different hyperparameters, different game sizes, different games, and different AlphaZero implementations. We would like to see if our results generalize to these other settings.

Exploring Our Hypotheses In Section 5.7.1, we hypothesize that game length is the key factor in explaining TD(λ) convergence speed and final model quality. Future work can explore this hypothesis to confirm or refute it. One inexpensive analysis is to look at the accuracy of the value targets over distance from the end of the game, over training iterations. We can use accuracy to approximate the amount of bias in TD(0) and the amount of variance in TD(1). This would provide empirical evidence for our hypothesis.

Improving Our Methods AlphaZero with TD(λ) achieves a higher quality final model than regular AlphaZero. TD(λ) interpolation retains TD(1)'s convergence speed. One direction for future work is to improve our method to achieve both a higher quality final model and faster convergence speed.

TD(λ) with refreshes is a promising direction to surpass the convergence speed of TD(1). However, it introduces instability in training. The problem of learning a nonstationary data distribution with NNs is a well-known problem in RL [3]. Future work attempting to stabilize TD(λ) with refreshes may have to solve this fundamental problem with NNs, because existing solutions (e.g., using replay buffers and target networks [64]) gain stability at the cost of slower convergence. Another option is to use a different function approximator more robust than NNs to nonstationarity, such as linear function approximation.

Our hypothesis above states that TD(1) converges to a worse final model than TD(0) because of a combination of TD(1)'s high variance targets and behaviour policy

churn. This suggests that, to get TD(1)'s final model to converge to TD(0)'s final model quality, requires many more samples per iteration.

What about learning a model better than TD(0)? Our method likely converges to the optimal behaviour policy, which accounts for the exploration noise we inject to induce exploration [9]. To converge to the true optimal policy, there are a few promising directions. One direction is to remove or anneal this exploration noise over training. Another direction is to correct for the exploration noise using Importance Sampling or Tree-Backup(λ) [3] (TB(λ) is a generalization of Q-learning applied to TD(λ)).

It is possible that AlphaZero with Tree-Backup(λ) and stable refreshes could learn an optimal policy with a faster convergence speed than regular AlphaZero. This is a promising direction for future work.

5.7.3 Related Work

There is previous work on TD(λ) in AlphaZero, but it is limited. KataGo uses auxiliary TD(λ)-like targets, and found that "neural nets seem to train slightly faster and achieve slightly better value and score loss" with them [65]. MuZero uses n-step TD targets ($n = 10$) when learning Atari games [40]. Deep Q-Networks (DQN) is another notable example of TD methods in Deep RL, using Q-learning [64].

The slow credit assignment problem in TD methods is a well-known challenge in RL, with existing work to address it. TD(λ) itself is one such approach, where λ can be tuned to balance between the slow one-step TD methods and MC methods [66]. Prioritized sweeping and prioritized experience replay are other approaches where the model prioritizes learning from more informative states or samples [67, 68]. Episodic backward updates is another approach that propagates the value of a state to the states that preceded it in the episode [69]. MuZero's Reanalyse method is another

approach that recalculates the value targets of states using the latest model [70]. Our TD(λ) with refreshes method is similar, but Reanalyse updates n-step targets using MCTS to generate the value estimate.

Chapter 6

Conclusion

In this thesis, we implemented AlphaZero on the game of Chinese Checkers, experimented with replacing AlphaZero's value targets with ground truth targets, and investigated the use of TD(λ) value targets.

We validated our AlphaZero implementation's learning ability with both win rate against a baseline and an increase in value head and policy head accuracy over training. We also measured the accuracy of the training data over training, finding that the accuracy of the value targets remained consistent over training, with a lower accuracy than the value head.

We replicated an experiment from prior work [16] which found that using ground truth targets in place of the default Monte Carlo value targets prevented learning. We found that this was due to majority-class collapse in the value head, and developed mitigation strategies to prevent this collapse. With these strategies in place, we found that ground truth targets enable the model to achieve a higher value head accuracy than the default Monte Carlo targets. The mitigation strategies also enabled us to experiment with other targets which have an imbalanced distribution, such as the TD(λ) targets.

We trained AlphaZero with TD(λ) value targets, including two additional settings "TD(λ) interpolation" and "TD(λ) with refreshes". We found TD(λ) value targets

yielded a higher value head accuracy in the 25-6 and 49-4 settings, a higher win rate in the 25-6, 36-6, and 49-4 settings, and higher value target accuracy. However, TD(λ) value targets also yielded slower convergence, especially in the larger boards, which we theorized was due to longer game lengths slowing credit assignment and increasing variance. We also suggest future work to strengthen our results, explore our hypotheses, and possibly further improve AlphaZero.

We hope that the insights in this thesis will help future researchers better understand AlphaZero and its training dynamics.

Bibliography

[1] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021.

[2] S. Legg and M. Hutter, "Universal Intelligence: A Definition of Machine Intelligence," *Minds and Machines*, vol. 17, no. 4, pp. 391–444, Nov. 30, 2007. DOI: 10.1007/s11023-007-9079-x.

[3] R. S. Sutton and A. Barto, *Reinforcement Learning: An Introduction* (Adaptive Computation and Machine Learning), Second edition. Cambridge, Massachusetts; London, England: The MIT Press, 2020, 526 pp.

[4] D. Silver *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, Dec. 7, 2018. DOI: 10.1126/science.aar6404.

[5] Y. Tian *et al.*, "ELF OpenGo: An analysis and open reimplementation of AlphaZero," in *Proceedings of the 36th International Conference on Machine Learning*, K. Chaudhuri and R. Salakhutdinov, Eds., ser. Proceedings of Machine Learning Research, vol. 97, PMLR, Jun. 9–15, 2019, pp. 6244–6253. [Online]. Available: https://proceedings.mlr.press/v97/tian19a.html.

[6] D. J. Wu, *Accelerating Self-Play Learning in Go*, 2020. arXiv: 1902.10565 [cs.LG].

[7] A. Trudeau and M. Bowling, "Targeted search control in AlphaZero for effective policy improvement," in *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, ser. AAMAS '23, London, United Kingdom: International Foundation for Autonomous Agents and Multiagent Systems, 2023, pp. 842–850.

[8] Y.-J. Tsai, W.-Y. Chen, Y.-R. Ju, Y.-H. Chang, and T.-R. Wu, "Regret-Guided Search Control for Efficient Learning in AlphaZero," in *The Fourteenth International Conference on Learning Representations*, 2026. [Online]. Available: https://openreview.net/forum?id=Eoiu5iJD71.

[9] D. Willemsen, H. Baier, and M. Kaisers, "Value targets in off-policy AlphaZero: A new greedy backup," *Neural Computing and Applications*, vol. 34, no. 3, pp. 1801–1814, Feb. 2022. DOI: 10.1007/s00521-021-05928-5.

[10] T. T. Wang *et al.*, "Adversarial Policies Beat Superhuman Go AIs," in *Proceedings of the 40th International Conference on Machine Learning*, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., ser. Proceedings of Machine Learning Research, vol. 202, PMLR, Jul. 23, 2023, pp. 35 655–35 739. [Online]. Available: https://proceedings.mlr.press/v202/wang23g.html.

[11] N. R. Sturtevant, "On Strongly Solving Chinese Checkers," in *Advances in Computer Games*, T. Cazenave, J. van den Herik, A. Saffidine, and I.-C. Wu, Eds., vol. 12516, Cham: Springer International Publishing, 2020, pp. 155–166. DOI: 10.1007/978-3-030-65883-0_13.

[12] A. Prasad. "Lessons from AlphaZero: Connect Four," Oracle Developers. (Jun. 20, 2018), [Online]. Available: https://medium.com/oracledevs/lessons-from-alphazero-connect-four-e4a0ae82af68.

[13] R. Haque, "On the Road to Perfection? Evaluating Leela Chess Zero Against Endgame Tablebases," M.S. thesis, University of Alberta, 2022. DOI: 10.7939/R3-5S4K-Q622.

[14] A. Husna, "Analyzing KataGo: A Comparative Evaluation Against Perfect Play in the Game of Go," M.S. thesis, University of Alberta, 2024. DOI: 10.7939/R3-E6QX-HS52.

[15] B. Karki, "Evaluating AlphaZero in a Strongly Solved Game," M.S. thesis, University of Alberta, 2024. DOI: 10.7939/R3-AB8H-XE50.

[16] B. Karki, Z. F. Ahmad, and N. R. Sturtevant, "Diagnosing Weaknesses in AlphaZero Learning," Unpublished manuscript, University of Alberta, 2024.

[17] L. V. Allis, "Searching for solutions in games and artificial intelligence," Ph.D. dissertation, Maastricht University, 1994. DOI: 10.26481/dis.19940923la.

[18] E. V. Nalimov, G. McC. Haworth, and E. Heinz, "Space-Efficient Indexing of Chess Endgame Tables," *ICGA Journal*, vol. 23, no. 3, pp. 148–162, Sep. 1, 2000. DOI: 10.3233/ICG-2000-23304.

[19] K. Thompson, "Retrograde Analysis of Certain Endgames," *ICGA Journal*, vol. 9, no. 3, pp. 131–139, Sep. 1, 1986. DOI: 10.3233/ICG-1986-9302.

[20] J. Schaeffer *et al.*, "Checkers Is Solved," *Science*, vol. 317, no. 5844, pp. 1518–1522, Sep. 14, 2007. DOI: 10.1126/science.1144079.

[21] L. V. Allis, H. J. van den Herik, and M. P. H. Huntjens, "Go-Moku Solved by New Search Techniques," *Computational Intelligence*, vol. 12, no. 1, pp. 7–23, Feb. 1996. DOI: 10.1111/j.1467-8640.1996.tb00250.x.

[22] H. Takizawa, *Othello is solved*, 2024. arXiv: 2310.19387 [cs.AI].

[23] O. Randall, M. Müller, T.-H. Wei, and R. Hayward, "Expected work search: Combining win rate and proof size estimation," in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, K. Larson, Ed., International Joint Conferences on Artificial Intelligence Organization, Aug. 2024, pp. 7003–7011. DOI: 10.24963/ijcai.2024/774.

[24] J. Romein and H. Bal, "Solving awari with parallel retrograde analysis," *Computer*, vol. 36, no. 10, pp. 26–33, Oct. 2003. DOI: 10.1109/MC.2003.1236468.

[25] G. Irving, *Pentago is a first player win: Strongly solving a game using parallel in-core retrograde analysis*, 2014. arXiv: 1404.0743 [cs.DC].

[26] L. V. Allis, "A Knowledge-Based Approach of Connect-Four: The Game Is Solved: White Wins," *ICGA Journal*, vol. 11, no. 4, pp. 165–165, Dec. 1, 1988. DOI: 10.3233/ICG-1988-11410.

[27] J. D. Allen, "A Note on the Computer Solution of Connect-Four," in *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad*, D. N. L. Levy and D. F. Beal, Eds., Chichester: Ellis Horwood, 1989, pp. 134–135.

[28] H. van den Herik, J. W. Uiterwijk, and J. van Rijswijck, "Games solved: Now and in the future," *Artificial Intelligence*, vol. 134, no. 1–2, pp. 277–311, Jan. 2002. DOI: 10.1016/S0004-3702(01)00152-7.

[29] R. de Man and B. Guo, *Syzygy Endgame Tablebases*, GitHub, 2013. [Online]. Available: https://github.com/syzygy1/tb.

[30] R. Hayward, Y. Björnsson, M. Johanson, M. Kan, N. Po, and J. van Rijswijck, "Solving 7×7 Hex with domination, fill-in, and virtual connections," *Theoretical Computer Science*, vol. 349, no. 2, pp. 123–139, Dec. 2005. DOI: 10.1016/j.tcs.2005.09.042.

[31] M. Müller, "Computer go as a sum of local games: An application of combinatorial game theory," Ph.D. dissertation, ETH Zurich, 1995, 109 pp. DOI: 10.3929/ETHZ-A-001440588. HDL: 20.500.11850/142058.

[32] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence*, vol. 6, no. 4, pp. 293–326, Win. 1975. DOI: 10.1016/0004-3702(75)90019-3.

[33] M. Campbell, A. Hoane, and F.-h. Hsu, "Deep Blue," *Artificial Intelligence*, vol. 134, no. 1–2, pp. 57–83, Jan. 2002. DOI: 10.1016/S0004-3702(01)00129-1.

[34] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 28, 2016. DOI: 10.1038/nature16961.

[35] C. B. Browne *et al.*, "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012. DOI: 10.1109/TCIAIG.2012.2186810.

[36] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *Computers and Games*, H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds., vol. 4630, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 72–83. DOI: 10.1007/978-3-540-75538-8_7.

[37] L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning," in *Machine Learning: ECML 2006*, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., vol. 4212, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293. DOI: 10.1007/11871842_29.

[38] C. D. Rosin, "Multi-armed bandits with episode context," *Annals of Mathematics and Artificial Intelligence*, vol. 61, no. 3, pp. 203–230, Mar. 2011. DOI: 10.1007/s10472-011-9258-6.

[39] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017. DOI: 10.1038/nature24270.

[40] J. Schrittwieser *et al.*, "Mastering Atari, Go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, Dec. 24, 2020. DOI: 10.1038/s41586-020-03051-4.

[41] D. Silver *et al.*, *Mastering chess and shogi by self-play with a general reinforcement learning algorithm*, 2017. arXiv: 1712.01815 [cs.AI].

[42] H. Yamashita and D. Dailey. "Computer Go Server (CGOS) 9x9 Standings." Archived at https://web.archive.org/web/20260812210258/http://www.yss-aya.com/cgos/9x9/standings.html. (2007), [Online]. Available: http://www.yss-aya.com/cgos/9x9/standings.html.

[43] H. Yamashita and D. Dailey. "Computer Go Server (CGOS) 13x13 Standings." Archived at https://web.archive.org/web/20260812210330/http://www.yss-aya.com/cgos/13x13/standings.html. (2007), [Online]. Available: http://www.yss-aya.com/cgos/13x13/standings.html.

[44] H. Yamashita and D. Dailey. "Computer Go Server (CGOS) 19x19 Standings." Archived at https://web.archive.org/web/20260812210347/http://www.yss-aya.com/cgos/19x19/standings.html. (2007), [Online]. Available: http://www.yss-aya.com/cgos/19x19/standings.html.

[45] The LCZero Authors. "Leela Chess Zero." (2018), [Online]. Available: https://lczero.org.

[46] G. Haworth and N. Hernandez, "TCEC15: The 15th Top Chess Engine Championship," *ICGA Journal*, vol. 41, no. 3, pp. 153–163, Nov. 7, 2019. DOI: 10.3233/ICG-190115.

[47] G. Haworth and N. Hernandez, "The 17th Top Chess Engine Championship, TCEC17," *ICGA Journal*, vol. 42, no. 2–3, pp. 181–191, Nov. 10, 2020. DOI: 10.3233/ICG-200152.

[48] "Top Chess Engine Championship," TCEC Archive: Past Seasons and Results. (2026), [Online]. Available: https://tcec-chess.com/.

[49] Y. Nasu, *NNUE: Efficiently Updatable Neural-Network-based Evaluation Functions for Computer Shogi*, 2018. [Online]. Available: https://github.com/ynasu87/nnue/tree/master.

[50] T.-R. Wu *et al.*, "MiniZero: Comparative Analysis of AlphaZero and MuZero on Go, Othello, and Atari Games," *IEEE Transactions on Games*, vol. 17, no. 1, pp. 125–137, Mar. 2025. DOI: 10.1109/TG.2024.3394900.

[51] A. E. Elo, *The Rating of Chessplayers, Past and Present*. New York: Arco Publishing, 1978.

[52] D. Balduzzi, K. Tuyls, J. Perolat, and T. Graepel, "Re-evaluating evaluation," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, (Montréal, Canada), ser. NIPS'18, Red Hook, NY, USA: Curran Associates Inc., 2018, pp. 3272–3283.

[53] O. Vinyals *et al.*, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, Nov. 14, 2019. DOI: 10.1038/s41586-019-1724-z.

[54] N. Natarajan, I. S. Dhillon, P. K. Ravikumar, and A. Tewari, "Learning with noisy labels," in *Advances in Neural Information Processing Systems*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds., vol. 26, Curran Associates, Inc., 2013. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2013/file/3871bd64012152bfb53fdf04b401193f-Paper.pdf.

[55] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 9, pp. 1263–1284, 2009. DOI: 10.1109/TKDE.2008.239.

[56] M. Buda, A. Maki, and M. A. Mazurowski, "A systematic study of the class imbalance problem in convolutional neural networks," *Neural Networks*, vol. 106, pp. 249–259, 2018. DOI: 10.1016/j.neunet.2018.07.011.

[57] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, Jun. 1, 2002. DOI: 10.1613/jair.953.

[58] R.-Y. Sun, "Optimization for Deep Learning: An Overview," *Journal of the Operations Research Society of China*, vol. 8, no. 2, pp. 249–294, Jun. 1, 2020. DOI: 10.1007/s40305-020-00309-6.

[59] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proceedings of the 30th International Conference on Machine Learning*, S. Dasgupta and D. McAllester, Eds., ser. Proceedings of Machine Learning Research, vol. 28, Atlanta, Georgia, USA: PMLR, Jun. 17–19, 2013, pp. 1310–1318. [Online]. Available: https://proceedings.mlr.press/v28/pascanu13.html.

[60] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Y. W. Teh and M. Titterington, Eds., ser. Proceedings of Machine Learning Research, vol. 9, Chia Laguna Resort, Sardinia, Italy: PMLR, May 13–15, 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a.html.

[61] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning*, F. Bach and D. Blei, Eds., ser. Proceedings of Machine Learning Research, vol. 37, Lille, France: PMLR, Jul. 7–9, 2015, pp. 448–456. [Online]. Available: https://proceedings.mlr.press/v37/ioffe15.html.

[62] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA: IEEE, Jun. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[63] S. Geman, E. Bienenstock, and R. Doursat, "Neural networks and the bias/-variance dilemma," *Neural Computation*, vol. 4, no. 1, pp. 1–58, 1992. DOI: 10.1162/neco.1992.4.1.1.

[64] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 26, 2015. DOI: 10.1038/nature14236.

[65] D. J. Wu. "Other Methods Implemented in KataGo." (Jun. 19, 2023), [Online]. Available: https://github.com/lightvector/KataGo/blob/master/docs/KataGoMethods.md#short-term-value-and-score-targets.

[66] E. Elelimy, B. Daley, A. Patterson, M. C. Machado, A. White, and M. White, "Deep reinforcement learning with gradient eligibility traces," in *Reinforcement Learning Conference*, 2025. [Online]. Available: https://openreview.net/forum?id=LZAafvwVMa.

[67] A. W. Moore and C. G. Atkeson, "Prioritized sweeping: Reinforcement learning with less data and less time," *Machine Learning*, vol. 13, no. 1, pp. 103–130, Oct. 1993. DOI: 10.1007/BF00993104.

[68] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1511.05952.

[69] S. Y. Lee, C. Sungik, and S.-Y. Chung, "Sample-efficient deep reinforcement learning via episodic backward update," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/e6d8545daa42d5ced125a4bf747b3688-Paper.pdf.

[70] J. Schrittwieser, T. Hubert, A. Mandhane, M. Barekatain, I. Antonoglou, and D. Silver, "Online and offline reinforcement learning by planning with a learned model," in *Proceedings of the 35th International Conference on Neural Information Processing Systems*, ser. Nips '21, Red Hook, NY, USA: Curran Associates Inc., 2021.

Appendix A

Chinese Checkers Statistics

This appendix provides useful reference statistics on the solve data of Chinese Checkers.

A.1 Formulas

The formula for number of states in the state space is given by

$$\begin{aligned} 2 \cdot \binom{N}{p}\binom{N-p}{p} &= 2 \cdot \frac{N!}{p!\,(N-p)!} \cdot \frac{(N-p)!}{p!\,(N-2p)!} \\ &= 2 \cdot \frac{N!}{p!\,p!\,(N-2p)!} \end{aligned} \tag{A.1}$$

where N is the number of locations in the board (e.g. $5^2 = 25$ for 25-6) and p is the number of pieces per player. The formulas for the number of terminal states and number of illegal states would be a useful derivation for future work.

The number of single-player states is given by

$$\binom{N}{p} \tag{A.2}$$

A.2 Tables

Solve data statistics for each board setting are reported in Table A.1. The win and loss distribution over the states with P1 to move is shown in Table A.2. The single-player statistics, used in the DBEval player, are in Table A.3.

Setting	States	Size (GB)	Wins	Draws	Illegal
16-3	320,320	8.0×10^{-5}	154,726 (48.30%)	0 (0.00%)	10,868 (3.39%)
25-6	9,610,154,400	2.4	4,749,618,788 (49.42%)	47,056,118 (0.49%)	63,860,706 (0.66%)
36-6	2,313,100,389,600	578.3	1,153,000,938,173 (49.85%)	5,199,820,604 (0.22%)	1,898,692,650 (0.08%)
49-4	63,136,929,240	15.8	31,532,340,944 (49.94%)	51,686,042 (0.08%)	20,561,310 (0.03%)

Table A.1: Solve data statistics for each board setting. The number of states includes the player to move. Wins are in the perspective of the maximizing player (P1). Losses omitted because they equal wins by symmetry. Size assumes two bits per state with no compression through symmetries. Data from Sturtevant [11].

Setting	States (P1 to move)	Wins	Losses	Draws	Illegal
16-3	160,160	96,875 (60.49%)	57,851 (36.12%)	0 (0.00%)	5,434 (3.39%)
25-6	4,805,077,200	3,023,861,831 (62.93%)	1,725,650,725 (35.91%)	23,478,135 (0.49%)	32,086,509 (0.67%)
49-4	31,568,464,620	18,224,424,980 (57.73%)	13,307,915,964 (42.16%)	25,843,021 (0.08%)	10,280,655 (0.03%)

Table A.2: Win, loss, draw, and illegal state counts over the states with P1 to move, which is half of the full state count. Wins and losses are in the perspective of the player to move (P1). The P2-to-move distribution is symmetric.

Setting	States	Size (MB)	Max depth	Depth at start
16-3	560	1.1×10^{-3}	8	7
25-6	177,100	0.35	12	11
36-6	1,947,792	3.90	17	15
49-4	211,876	0.42	18	13

Table A.3: Single-player statistics for each board setting (where only one player's pieces are on the board). Size assumes 16 bits per state to store the exact single-player distance to the goal. Max depth is the largest single-player distance in the database, and depth at start is the single-player distance from the starting position.

Appendix B

AlphaZero on Chinese Checkers

This appendix discusses alternative evaluation methods, reports the hyperparameters used for AlphaZero, and the number of parameters in the neural network.

B.1 Alternative Evaluation Methods

B.1.1 Targets

We use v_* and π_* as the ground truth value and policy targets, respectively, throughout the thesis. However, there are other possible ground truth targets we could use. We discuss two alternative targets, $v_{*\gamma}$ and $\pi_{*\gamma}$, below.

A possible direction for future work is to explore alternative value head targets. One desirable property of an alternative target is the ability to differentiate between states that are equivalent under v_*. For example, if the solve data contains DTW information, we can use a discounted value head target $v_{*\gamma}(s) = \gamma^{d_*(s)} v_*(s)$, where $\gamma \in [0, 1]$ is the discount factor and $d_*(s)$ is the distance-to-win. $v_{*\gamma}$ accounts for the distance to win under optimal play. This could enable different evaluation metrics to be used with the value head.

A possible direction for future work is to explore alternative policy head targets. One desirable property of an alternative target is the ability to differentiate between optimal actions. This is especially useful for policies on losing states. On winning

states, just the uniform optimal policy π_* is sufficient to win. Differentiating between winning actions is only necessary when you have a goal beyond winning, like winning in less moves. However, in losing states, differentiation can be the difference between winning and losing. Some options to differentiate between optimal actions include choosing actions that lead to a slower loss or choosing actions leading to states where the opponent has fewer optimal actions. Implicitly, both of these options attempt to maximize the probability of winning by increasing the chance of the opponent making a mistake.

For example, if the solve data contains DTW information, we could use $v_{*\gamma}$ to define a policy $\pi_{*\gamma}$ that prefers faster wins among winning actions and slower losses among losing actions. The probability distribution could be created with softmax or argmax, where argmax would assign all probability to the fastest winning action. This could enable different evaluation metrics to be used with the policy head.

B.1.2 Evaluation Metrics

We use accuracy as the primary evaluation metric for both the value and policy heads throughout the thesis. There are many other evaluation metrics we could use. For the value head, Mean Squared Error (MSE) and Mean Absolute Error (MAE) also capture the magnitude of the model's error which accuracy does not. MSE and MAE also extend to continuous value head targets.

The policy head has more options. Top-k accuracy checks whether any of the model's k most preferred actions is optimal, which may reflect the policy head's utility as a search prior in MCTS (optimal actions should be among those highly ranked by the policy). Another metric, optimal probability mass, measures the policy's quality when acting by sampling. It measures how much probability mass of the model's policy is placed on optimal actions. Equivalently, the probability that an action sampled

from $\hat{\pi}$ is optimal. The policy accuracy metric we use reflects the model's utility when acting greedily (taking the action with the highest probability).

Further, using alternate ground truth policies motivates more evaluation options. For example, using a policy that differentiates between optimal actions, like $\pi_{*\gamma}$, motivates metrics that capture how well the model's predicted distribution aligns with the ground truth distribution. This includes metrics like KL divergence ($D_{\mathrm{KL}}(\pi_* \parallel \hat{\pi})$) and cross entropy ($H(\pi_*, \hat{\pi})$).

Alternate ground truth policies also affect our accuracy metrics. We can generalize the set of optimal actions to any policy π by selecting the set of actions the policy most prefers:

$$A^*_\pi(s) = \{a \in A(s) : \pi(a|s) = \max_{b \in A(s)} \pi(b|s)\} \tag{B.1}$$

Under this definition, $A^*(s) = A^*_{\pi_*}(s)$. Accuracy against a ground truth policy like $\pi_{*\gamma}$ becomes stricter than accuracy against π_* as it requires the model to predict a smaller set of optimal actions.

B.1.3 State Set Filters

A possible direction for future work is to generalize the trivial state concern described in Section 3.1.1.3 by considering each state's proportion of optimal actions, $\rho(s) = |A^*(s)|/|A(s)|$ (for trivial states, $\rho(s) = 1$). For example, consider a dataset where $\rho(s) = 0.9$ for all states and one where $\rho(s) = 0.1$. A policy will naturally have a higher policy accuracy on the first dataset, and this higher accuracy does not reflect the model's ability.

To solve this, we can standardize this distribution. For example, we can create a uniform distribution over buckets of $\rho(s)$ by subsampling states from majority buckets until uniform. This ensures that the model's accuracy is not affected by

the distribution of $\rho(s)$ in the dataset. However, ensuring a standard distribution through subsampling is sometimes impossible, as some datasets will have no states in a particular bucket. An alternative is to calculate the accuracy of a uniformly random policy on the dataset (the baseline accuracy, $\mathbb{E}_{s\sim S}[\rho(s)]$), then report the model's accuracy relative to this baseline.

We could also report model accuracy on each ρ bucket in the dataset. This is more informative than accuracy, but also requires reporting more numbers.

B.1.4 Baseline Players

There are many other options we considered for the baseline player. Within MCTS, we explored different rollout depths, rollout policies, and evaluation functions. Two simple rollout policies are to choose a random action or a random forwards action. Another rollout policy chooses the action that moves the piece furthest from the goal as forwards as possible. For evaluation functions, a common one is to use the real reward at the terminal state (MCTS originally performed random rollouts to terminal for evaluation [36]). This requires a rollout to the end of the game.

Another evaluation function is DBEval. This function requires a pre-computed database holding the exact single-player distance to the goal for every possible piece configuration. The single-player distance tells you how many actions are required for the player to get all its pieces in the goal area, assuming no other pieces are on the board. This is the value DistEval approximates by summing each piece's distance to the goal. To calculate each state's value, DBEval follows the same procedure as DistEval in Equation (3.8), but replaces $\sum_{p\in P_i(s)} d(p, G_i)$ with the exact single-player distance. Computing this database is inexpensive for most board sizes, and small enough to store in memory. For example, 16-3 requires storing the distances for 560 states, and 25-6 177,100 (compared to 320K and 9.6B states in their entire state

spaces, respectively; see Tables A.1 and A.3 for details). We observe that DBEval performs similarly to DistEval, and only outperforms it on MCTS sample sizes larger than our settings (in the tens of thousands).

There are other players available, such as a random player or a minimax player, that could be used as a baseline in future work.

B.1.5 Player Evaluation Metrics

We use WLD rates in the thesis. Another possible metric is the average return: the mean game outcome $z \in \{-1, 0, 1\}$ over all games played. This summarizes the player's performance into a single number on the same $[-1, 1]$ scale as the game's return. However, it loses information about the player's performance compared to win/loss/draw rates.

An important aspect of the baseline player is stochasticity. The AlphaZero player is largely deterministic (PUCT with greedy sampling) during evaluation, so a deterministic baseline player leads to the same game being repeated. Two deterministic players can be compared by starting them from a random state.

B.2 Hyperparameters

Table B.1 reports the hyperparameters used for supervised learning, including the network architecture. Table B.2 reports the hyperparameters used for AlphaZero training and self-play. Table B.3 reports the hyperparameters used when evaluating a model as a player, and Table B.4 the hyperparameters used for the baseline player it is evaluated against. No formal hyperparameter search was performed. Some of these hyperparameters were chosen based on prior work, and others were chosen based on experience while implementing and testing AlphaZero.

Hyperparameter	Value
Batch size	256
Learning rate	5×10^{-5}
Weight decay	1×10^{-4}
Number of filters	256
Number of residual blocks	3

Table B.1: Hyperparameters used for supervised learning, covering both the network architecture and the optimizer.

Hyperparameter	Value
Samples per iteration	5,000
Replay buffer size	10,000
MCTS samples	512
Selection policy	PUCT
Rollout policy	Random
Rollout depth	8
Dirichlet noise ϵ	0.25
c_{puct}	1.0
Action selection	Sampled from π

Table B.2: Hyperparameters used for AlphaZero training and self-play. The supervised learning hyperparameters in Table B.1 also apply to the training step.

Hyperparameter	Value
MCTS samples	512
Rollout type	None
c_{puct}	0.25
Action selection	$\arg\max \pi$

Table B.3: Hyperparameters used when evaluating a model as a player. No rollouts are used, so leaf nodes are evaluated by the value head alone. Actions are selected greedily rather than sampled as in self-play (Table B.2), so the player is deterministic up to ties.

Hyperparameter	Value
MCTS samples	512
Selection policy	UCT
Rollout policy	Most forward
Rollout depth	8
Evaluation function	DistEval
Rollout ϵ	0.1
c	$\frac{1}{4} \cdot \frac{2n(b-1)}{\sqrt{2}}$

Table B.4: Hyperparameters used for the baseline player. ϵ is the probability that the rollout policy plays a random forward action instead of the most forward one. c is the UCT exploration constant, anchored to the half-range of DistEval, where n is the number of pieces each player has and b is the board size. This gives $c = 3.18$ for 16-3, 8.49 for 25-6, 10.61 for 36-6, and 8.49 for 49-4.

B.3 Neural Network Parameters

Table B.5 reports the number of parameters in the neural network for each board size, using the architecture in Table B.1.

Board	Cells	Parameters	Size (MB)
4×4	16	5,711,105	22.8
5×5	25	9,252,722	37.0
6×6	36	17,918,225	71.7
7×7	49	36,945,506	147.8

Table B.5: Neural network parameter counts for each board size, using the architecture in Table B.1. Size assumes four bytes per parameter (fp32). The number of pieces each player has does not affect the number of parameters.

B.4 Additional Results

B.4.1 Effect of Model Size on Accuracy

In the thesis we measured the effect of dataset size on model accuracy. We also measured the effect of model size (number of residual blocks) on model accuracy. Our results show that the model size (number of residual blocks) does not seem to increase final accuracy. These results are shown in Figure B.1.

B.4.2 Self-Play Statistics

Some statistics about the self-play data generated during AlphaZero training are shown in Table B.6.

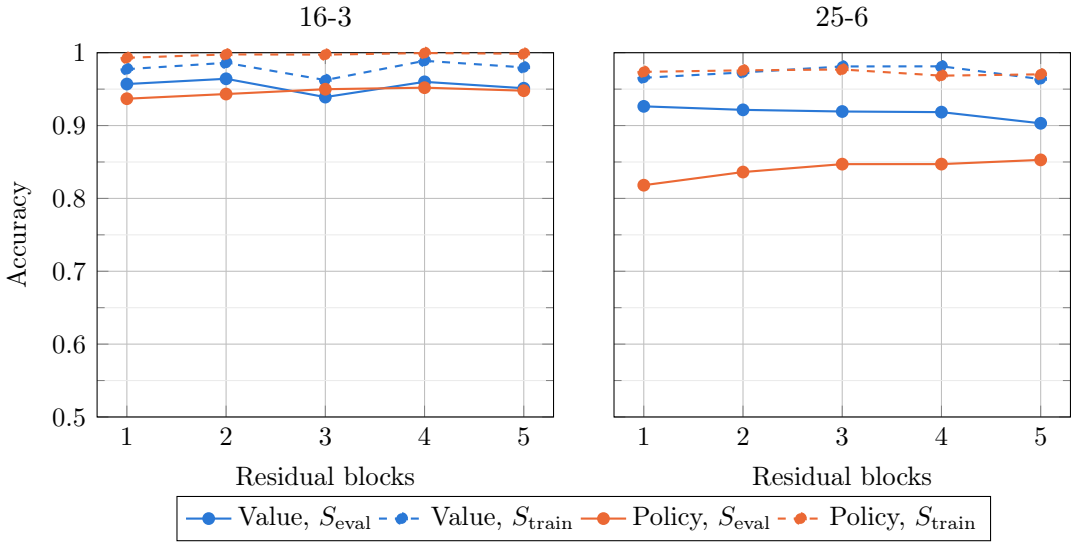


Figure B.1: Final accuracy after ten epochs of supervised learning versus model size, on 16-3 (left) and 25-6 (right), with a training set of 65,536 states. Model size is the number of residual blocks. Accuracy is measured on the S_{train} and S_{eval} state sets.

				Unique states		P1 rate		
Board	Iters	Turns/game	Games/iter	Count	Fraction	Win	Loss	Draw
16-3	148	17.9	280	150,425	25.1%	56.7%	43.3%	0.0%
25-6	200	30.7	164	741,389	73.9%	53.8%	46.2%	0.0%
36-6	200	44.9	113	832,094	82.8%	53.0%	47.0%	0.0%
49-4	200	43.2	118	774,737	77.1%	58.1%	41.9%	0.0%

Table B.6: Self-play statistics for each board size, averaged over training iterations and trials. Every board is given the same per-iteration state budget of about 5,000 states, so the number of games per iteration falls as games get longer. The unique-state count is the number of distinct states across all of self-play, and the fraction is that count divided by the total number of states generated. Self-play games essentially never draw: the highest draw rate on any board at any iteration is 0.02%.

Appendix C

AlphaZero with TD(λ)

In this section, we present some extra results from the TD(λ) experiments. We include additional figures for policy head accuracy, policy target accuracy, and a hyperparameter sweep.

C.1 Policy Head Accuracy

Board	States	$\lambda < 1$	$\lambda = 1$	TD(1) difference
16-3	Seen	93.9–94.3	94.5	+0.5 to +0.2
	Random	90.7–90.9	90.6	−0.2 to −0.4
25-6	Seen	86.4–86.9	86.4	0.0 to −0.5
	Random	85.9–86.4	85.5	−0.4 to −0.8
49-4	Seen	87.4–90.1	89.3	+1.9 to −0.8
	Random	83.4–86.3	86.4	+3.0 to 0.0

Table C.1: Converged policy head accuracy (%), averaged over the last 20% of training iterations. Columns, the omission of the 36-6 setting, and the seed counts are as in Table 5.1.

The policy head accuracy results for the TD(λ) experiments in the 16-3, 25-6, and 49-4 settings are shown in Figures C.1 to C.3. The converged policy head accuracy

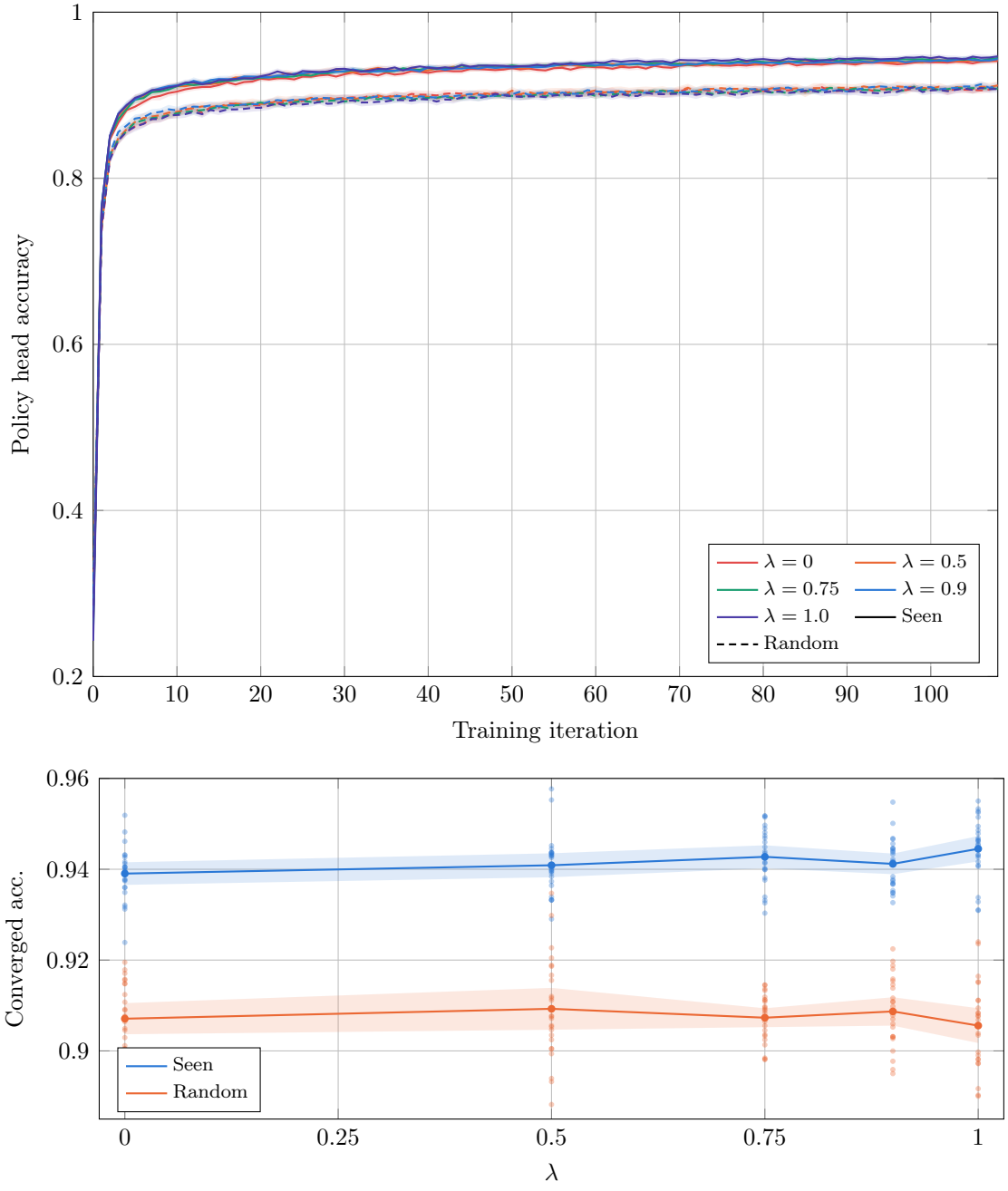


Figure C.1: Policy head accuracy over training on 16-3 across λ values, on seen (solid) and random (dashed) states with shaded 95% CI bands. Bottom: the converged (last-20%) accuracy over λ, with 95% CI bands and the per-trial converged values as dots. Averaged over 25 seeds.

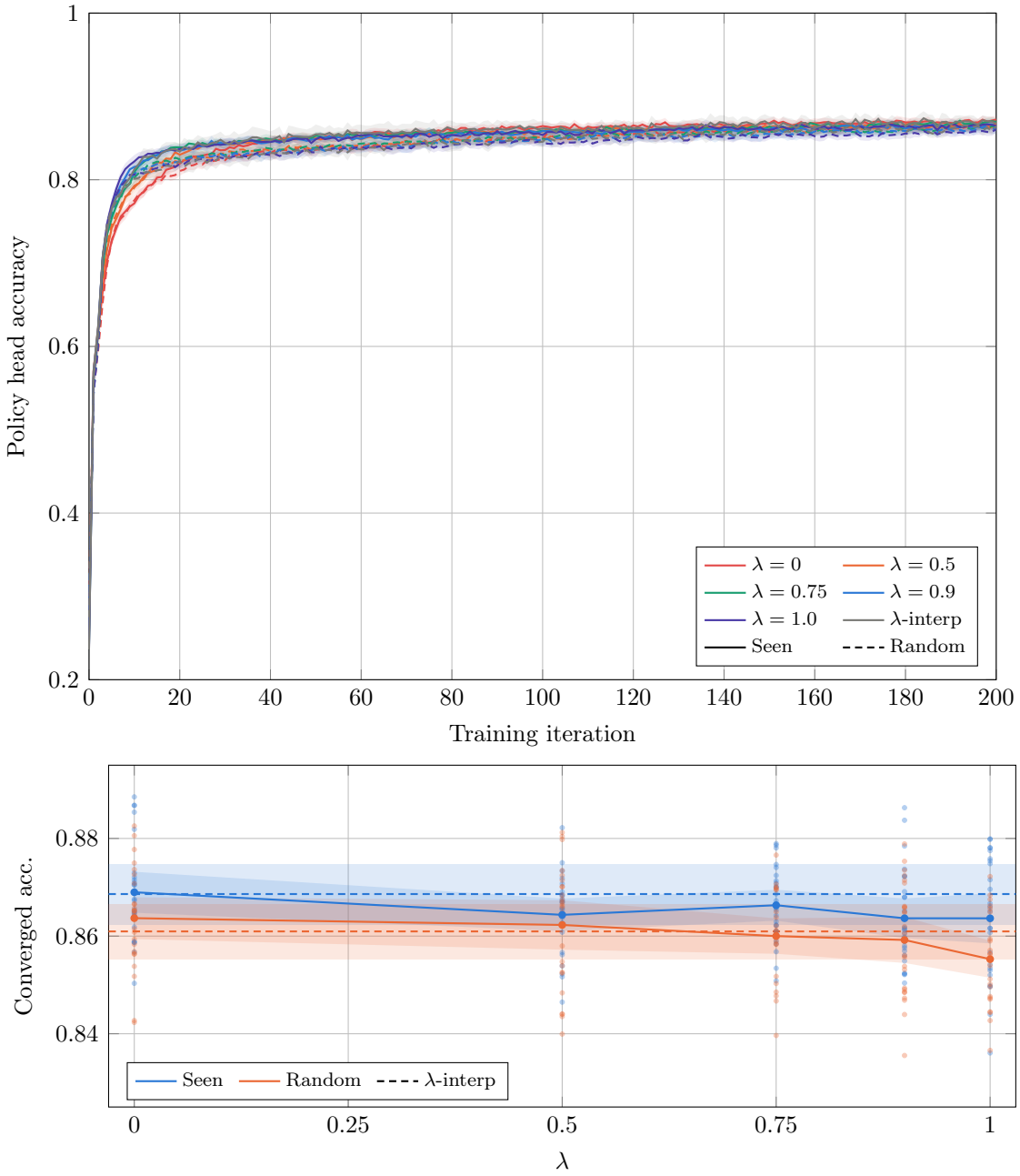


Figure C.2: Policy head accuracy over training on 25-6 across λ values (plus the lambda-interpolation run), on seen (solid) and random (dashed) states with shaded 95% CI bands. Bottom: the converged (last-20%) accuracy over λ, with 95% CI bands and the per-trial converged values as dots; dashed lines mark the lambda-interpolation run. Averaged over 25 seeds.

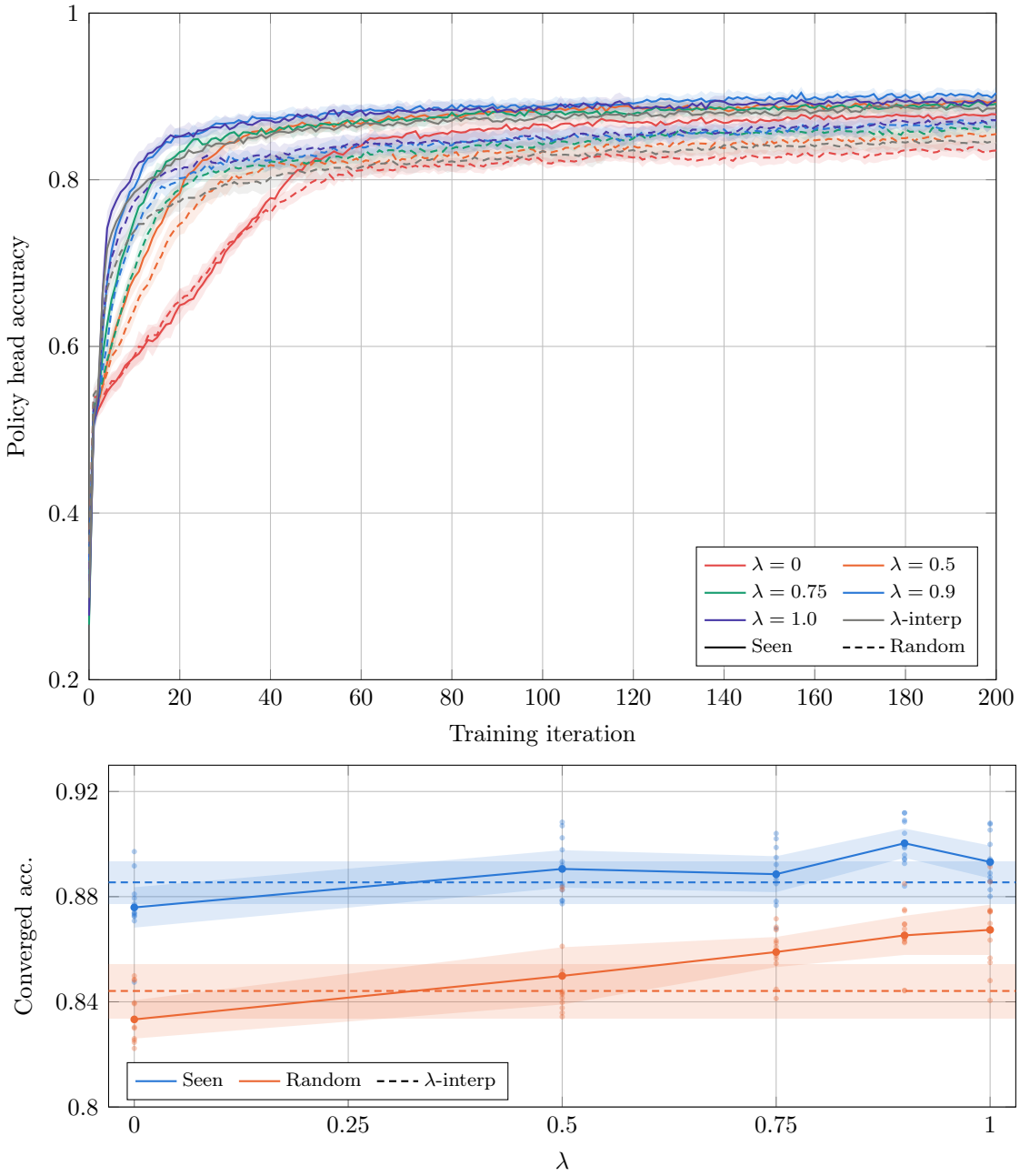


Figure C.3: Policy head accuracy over training on 49-4 across λ values (plus the lambda-interpolation run), on seen (solid) and random (dashed) states with shaded 95% CI bands. Bottom: the converged (last-20%) accuracy against λ, with 95% CI bands and the per-trial converged values as dots; dashed lines mark the lambda-interpolation run. Averaged over 12 seeds.

for each setting is summarized in Table C.1.

C.2 Policy Target Accuracy

The policy target accuracy results for the TD(λ) experiments in the 16-3, 25-6, and 49-4 settings are shown in Figures C.4 to C.6.

C.3 Hyperparameter Sweep

To ensure that our results are not due to our specific hyperparameters favouring TD(λ), we performed a hyperparameter sweep for the TD(λ) experiments. We swept over learning rate, weight decay, c_{puct}, Dirichlet ϵ, and replay buffer size. The results of this sweep are shown in Figure C.7. We found that the trends observed in the main experiments held across a range of hyperparameters, with the final value head accuracy being higher for TD(λ) than for the default Monte Carlo targets.

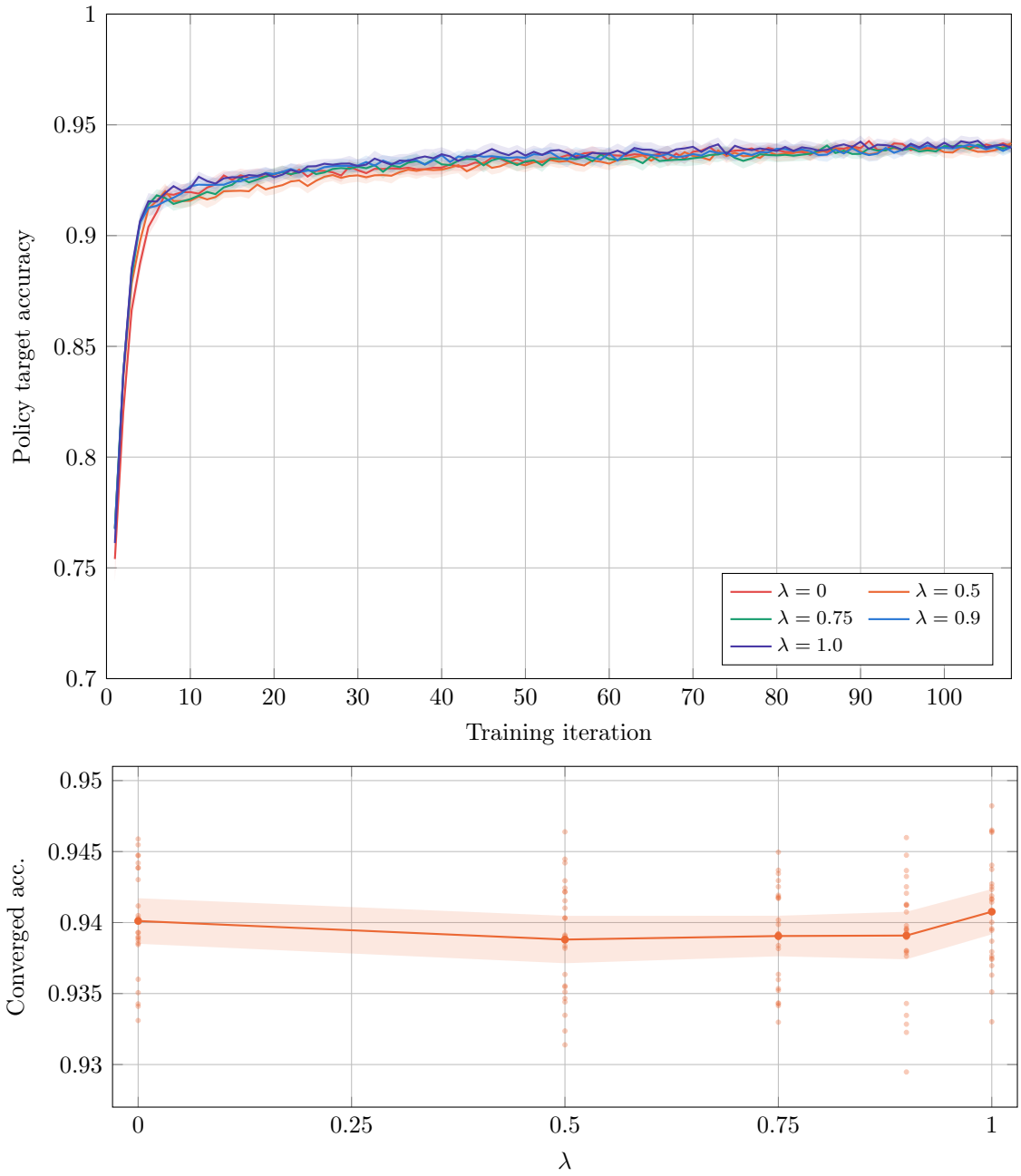


Figure C.4: Accuracy of the policy targets over training on 16-3, one line per λ with shaded 95% CI bands. Bottom: the converged (last-20%) accuracy over λ, with 95% CI bands and the per-trial converged values as dots. Averaged over 25 seeds.

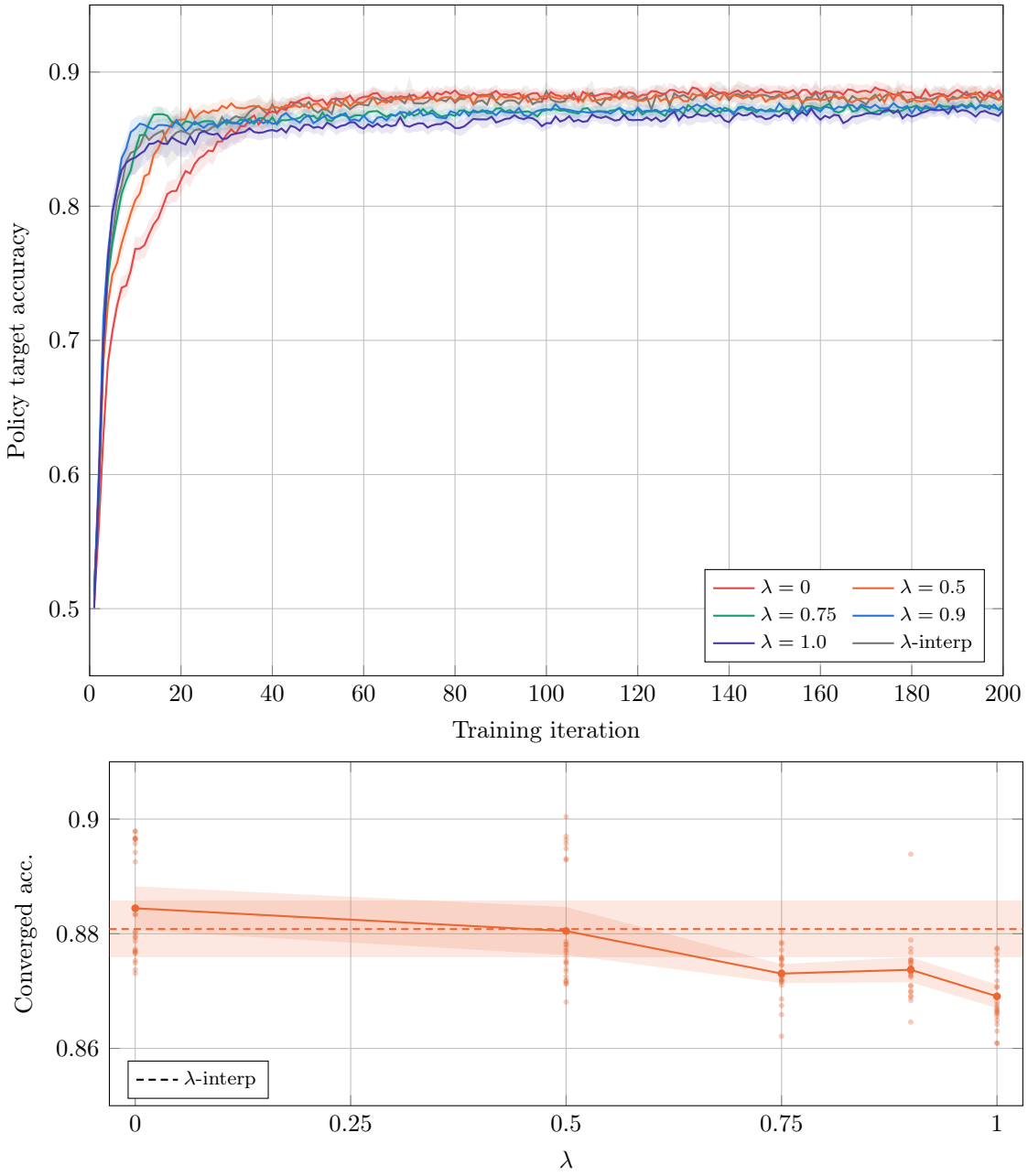


Figure C.5: Accuracy of the policy targets over training on 25-6, one line per λ (plus the lambda-interpolation setting) with shaded 95% CI bands. Bottom: the converged (last-20%) accuracy over λ, with 95% CI bands and the per-trial converged values as dots; the dashed line marks the lambda-interpolation setting. Averaged over 25 seeds.

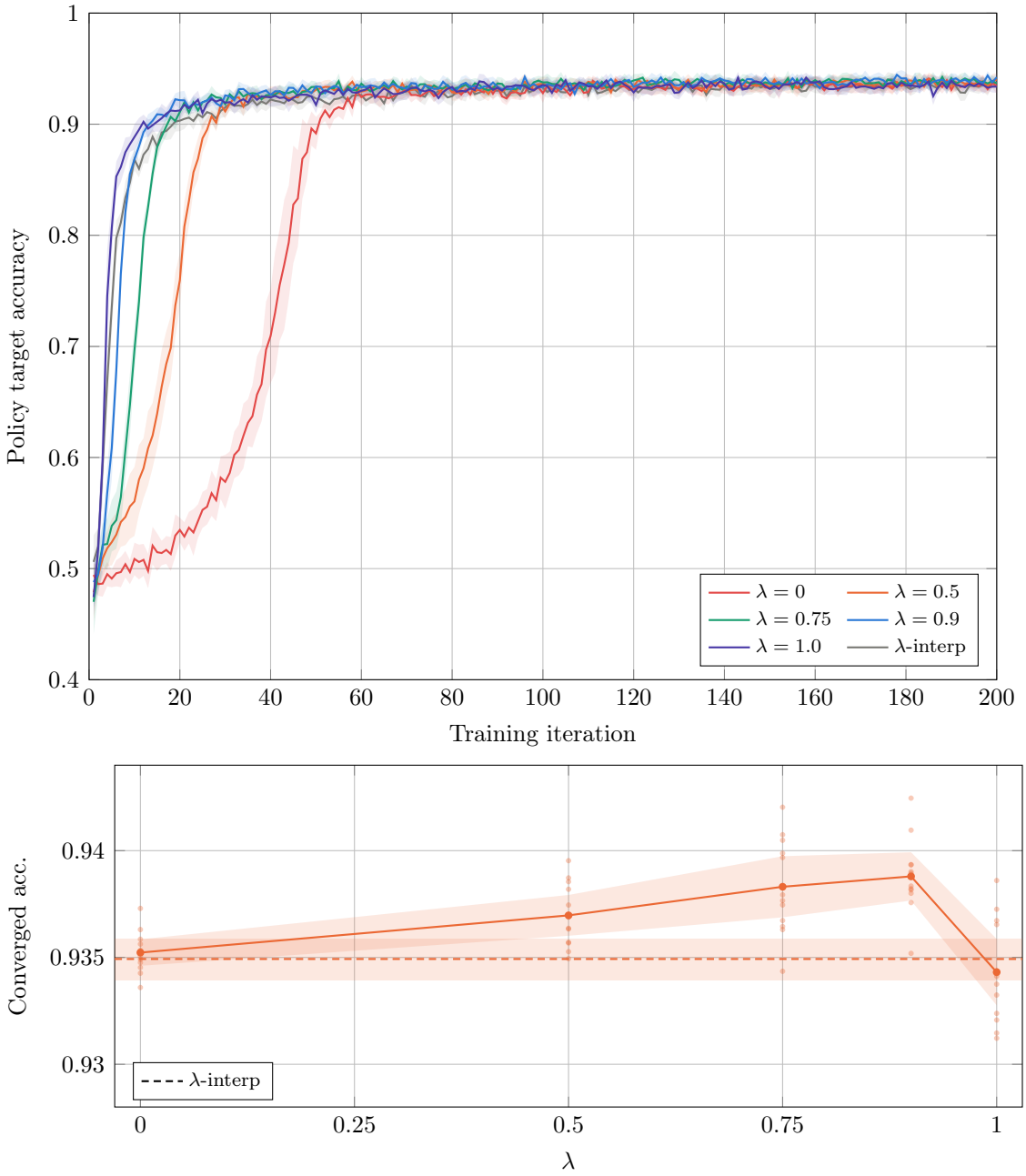


Figure C.6: Accuracy of the policy targets over training on 49-4, one line per λ (plus the lambda-interpolation setting) with shaded 95% CI bands. Policy targets come from MCTS visit counts and are unaffected by λ. Bottom: the converged (last-20%) accuracy over λ, with 95% CI bands and the per-trial converged values as dots; the dashed line marks the lambda-interpolation setting. Averaged over 12 seeds.

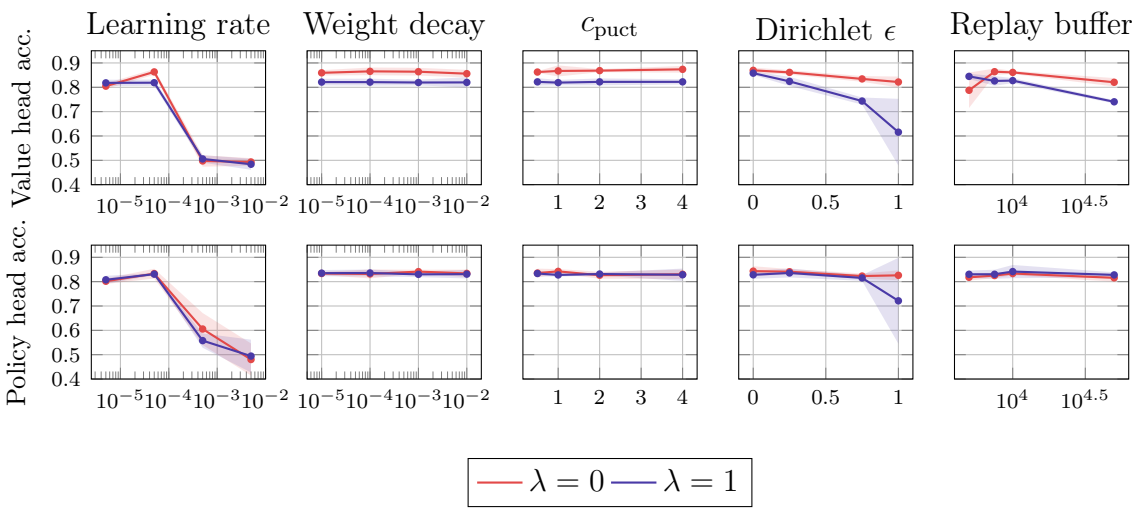


Figure C.7: Last-20% model accuracy on the random (held-out) set against five training hyperparameters, for the value (top) and policy (bottom) heads, at $\lambda = 0$ and $\lambda = 1$. Every configuration trained for 50 iterations; shaded bands are 95% CIs over 5 trials. The two λ settings track each other across the sweeps: both learn over a wide range and both collapse at large learning rates.