PLAYING NONDETERMINISTIC GAMES THROUGH PLANNING WITH A LEARNED MODEL

Anonymous authors
Paper under double-blind review

ABSTRACT

The MuZero algorithm is known for achieving high-level performance on traditional zero-sum two-player games of perfect information such as chess, Go, and shogi, as well as visual, non-zero sum, single-player environments such as the Atari suite. Despite lacking a perfect simulator and employing a learned model of environmental dynamics, MuZero produces game-playing agents comparable to its predecessor, AlphaZero. However, the current implementation of MuZero is restricted only to deterministic environments. This paper presents Nondeterministic MuZero (NDMZ), an extension of MuZero for nondeterministic, two-player, zero-sum games of perfect information. Borrowing from Nondeterministic Monte Carlo Tree Search and the theory of extensive-form games, NDMZ formalizes chance as a player in the game and incorporates the chance player into the MuZero network architecture and tree search. Experiments show that NDMZ is capable of learning effective strategies and an accurate model of the game.

1 INTRODUCTION

While the AlphaZero algorithm achieved superhuman performance in a variety of challenging domains, it relies upon a perfect simulation of the environment dynamics to perform precision planning. MuZero, the newest member of the AlphaZero family, combines the advantages of planning with a learned model of its environment, allowing it to tackle problems such as the Atari suite without the advantage of a simulator. This paper presents Nondeterministic MuZero (NDMZ), an extension of MuZero to stochastic, two-player, zero-sum games of perfect information. In it, we formalize the element of chance as a player in the game, determine a policy for the chance player via interaction with the environment, and augment the tree search to allow for chance actions.

As with MuZero, NDMZ is trained end-to-end in terms of policy and value. However, NDMZ aims to learn two additional quantities: the player identity policy and the chance player policy. With the assumption of perfect information, we change the MuZero architecture to allow agents to determine whose turn it is to move and when a chance action should occur. The agent learns the fixed distribution of chance actions for any given state, allowing it to model the effects of chance in the larger context of environmental dynamics. Finally, we introduce new node classes to the Monte Carlo Tree Search, allowing the search to accommodate chance in a flexible manner. Our experiments using Nannon, a simplified variant of backgammon, show that NDMZ approaches AlphaZero in terms of performance and attains a high degree of dynamics accuracy.

2 PRIOR WORK

2.1 EARLY TREE SEARCH

The classic search algorithm for perfect information games with chance events is expectiminimax, which augments the well-known minimax algorithm with the addition of of chance nodes (Mitchie, 1966; Maschler et al., 2013). Chance nodes assume the expected value of a random event taking place, taking a weighted average of each of its children, based on the probability of reaching the child, rather than the max or min. Chance nodes, min nodes, and max nodes are interleaved depending on the game being played. The *-minimax algorithm extends the alpha-beta tree pruning strategy to games with chance nodes (Ballard, 1983; Knuth & Moore, 1975).

2.2 MODEL-FREE REINFORCEMENT LEARNING

A notable success of model-free reinforcement learning for nondeterministic games is TD-Gammon, an adaptation of Sutton & Barto (2018)'s TD-Lambda algorithm to the domain of backgammon (Tesauro, 1995). Using temporal difference learning, a network is trained to produce the estimated value of a state. When it is time to make a move, each legal move is used to produce a successor state, and the move with the maximum expected outcome is chosen. Agents produced by TD-Gammon were able to compete with world-champion backgammon players, although it was argued that the stochastic nature of backgammon assisted the learning process (Pollack & Blair, 1996).

2.3 MONTE CARLO TREE SEARCH

The Monte Carlo Tree Search (MCTS) algorithm attracted interest after its success with challenging deterministic problems such as the game Go, and it has been adapted for stochastic games in an number of different ways (Coulom, 2006; Browne et al., 2012). A version of MCTS by Van Lishout et al. (2007) called called McGammon (Monte Carlo Backgammon) was capable of finding expert moves in a simplified backgammon game. At a chance node, the next move is always chosen randomly; at choice nodes, the Upper Confidence bound applied to Trees (UCT) algorithm is used in the selection phase to choose player actions (Kocsis & Szepesvári, 2006).

The McGammon approach was adapted by Yen et al. (2014) to the game of Chinese Dark Chess and formalized as Nondeterministic Monte Carlo Tree Search (NMCTS). There are two node types in NMCTS: deterministic state nodes and nondeterministic state nodes. Nondeterministic state nodes contain at least one inner node; each inner node has its own visit count, win count, and subtree. Upon reaching a nondeterministic state node, an inner node is chosen using "roulette wheel selection" based on the probability distribution of possible resulting states. NMCTS was designed to suit the nature of Chinese Dark Chess, in which both types of state nodes may exist on the same level of the search tree; this is in contrast to backgammon, in which homogeneous layers of chance nodes and choice nodes are predictably interleaved.

Alternative MCTS approaches have been adapted for densely stochastic games, wherein the the branching factor at chance nodes is so great that successor states at chance nodes are unlikely to ever be resampled. These include Double Progressive Widening and Monte Carlo *-Minimax Search (Chaslot et al.; Lanctot et al., 2013). In this paper, however, we are concerned with games that are not densely stochastic.

2.4 ALPHAZERO AND MUZERO

MCTS served as the foundation of the AlphaGo and AlphaZero algorithms, the latter of which attained superhuman performance in the deterministic domains of chess, Go, and shogi (Silver et al., 2016; b;a). AlphaZero substitutes neural network activations for random rollouts to produce policy and value evaluations, employs the PUCT algorithm during search, and trains solely via self-play reinforcement learning (Rosin, 2011). Hsueh et al. (2018) adapted AlphaZero for a simplified, solved version of the nondeterministic game of Chinese Dark Chess, finding that AlphaZero is capable of attaining optimal play in a stochastic setting.

A notable limitation of AlphaZero is that the agent relies upon a perfect simulator in order to perform its lookahead search. The MuZero algorithm frees AlphaZero of this restriction, creating a model of environmental dynamics as well as game-playing strategies through interactions with the environment and self-play (Schrittwieser et al., 2019). Until recently, the state of the art in complex single-player domains such as Atari games involved model-free reinforcement learning approaches such as Q-learning and the family of algorithms that followed (Mnih et al., 2015; Çalışır & Pehlivanoğlu, 2019). Despite the burden of learning enviornmental dynamics, MuZero leverages its precision-planning capabilities to achieve an extremely strong degree of performance on these problems.

Given parameters θ and a state representation at timestep k, AlphaZero employs a single neural network f_θ with a policy head and a value head: $\boldsymbol{p}^k, v^k = f_\theta(s^k)$. In contrast, the MuZero model employs three networks for representation, prediction, and dynamics. A set of observations is passed into the representation function h_θ to produce the initial hidden state: $s^0 = h_\theta((o_1, \ldots, o_t))$. The dynamics function recurrently takes a previous hidden state and an action image to produce an

immediate reward and the next hidden state: $r^k, s^k = g_\theta(s^{k-1}, a^k)$. Any hidden state produced in such fashion may be passed into the prediction function f_θ to produce a policy vector and scalar value: $\boldsymbol{p}^k, v^k = f_\theta(s^k)$.

With this model, the agent may search over future trajectories $a^1, ..., a^k$ using past observations $o_1, ..., o_t$; in principle any MDP planning algorithm may be used given the rewards and states produced by the dynamics function. MuZero uses a MCTS algorithm similar to AlphaZero's search, wherein the representation function is used to generate the hidden state for the root node. Each child node then has its own hidden state, produced by the dynamics function, and prior probability and value, produced by the prediction function. These values are used in the tree search to select an action $a_{t+1} \sim \pi_t$ via PUCT.

3 NONDETERMINISTIC MUZERO

3.1 DEFINITIONS

We consider here an extension of the MuZero algorithm for two-player, zero-sum games of perfect information with chance events. Following Lanctot et al. (2019), we will characterize such games as extensive-form games with the tuple $\langle \mathbb{N}, \mathbb{A}, \mathbb{H}, \mathbb{Z}, u, \iota, \mathbb{S} \rangle$, wherein

- $\mathbb{N} = \{1, 2, c\}$ includes the two rival players as well as the special **chance player** c;
- $\mathbb{A}$ is the finite set of actions that players may take;
- $\mathbb{H}$ is the set of histories, or sequence of actions taken from the start of the game;
- $\mathbb{Z} \subseteq \mathbb{H}$ is the subset of terminal histories;
- $u : \mathbb{Z} \to \Delta_u^n \subseteq \mathbb{R}^n$, where $\Delta_u = [u_{min}, u_{max}]$, is the utility function assigning each player a utility at terminal states;
- $\iota : \mathbb{H} \to \mathbb{N}$ is the player identity function, such that $\iota(h)$ identifies the player to act at history h;
- $\mathbb{S}$ is the set of states, in which $\mathbb{S}$ is a partition of $\mathbb{H}$ such that each state $s \in \mathbb{S}$ contains histories $h \in s$ that cannot be distinguished by $\iota(s) = \iota(h)$ where $h \in s$.
- The legal actions available at state s are denoted as $\mathbb{A}(s) \subseteq \mathbb{A}$;
- Zero-sum games are characterized such that $\forall z \in \mathbb{Z}, \sum_{i \in \mathbb{N}} u_i(z) = 0$; and
- A game of perfect information is one with only one history per state: $\forall s \in \mathbb{S}, |s| = 1$.

A **chance node** (or chance event) is a history h such that $\iota(h) = c$. A **policy** $\pi : \mathbb{S} \to \Delta(\mathbb{A}(s))$, where $\Delta(\mathbb{X})$ represents the set of probability distributions over $\mathbb{X}$, describes agent behavior. An agent acts by sampling an action from its policy: $a \sim \pi$. A **deterministic** policy is one where the distribution over actions has probability 1 on one action and zero on others. A policy that is not necessarily deterministic is called **stochastic**. The chance player always plays with a fixed, stochastic policy π_{chance}.

A **transition function** $\mathbb{T} : \mathbb{S} \times \mathbb{A} \to \Delta(\mathbb{S})$ defines a probability distribution over successor states s' when choosing action a from state s. Because states are simply sequences of previous actions, a transition function can equivalently be represented using intermediate chance nodes between the histories of the predecessor and successor states $h \in s$ and $h' \in s'$. The transition function is then determined by π_{chance}.

As we are concerned with perfect information games, we shall use s interchangeably with h throughout this text, such that s may refer to the single history h contained within it. For ease of notation we refer to the chance policy π_{chance} as ψ.

3.2 MODIFICATIONS

First, we shall extend $\mathbb{N}$ such that $\mathbb{N} = \{1, 2, c, d\}$ includes the new **terminal player** d. The terminal player is to act when a terminal state is reached, such that $\forall z \in \mathbb{Z}, \iota(z) = d$. We shall refer to nonchance, nonterminal players as **choice players**. Let us characterize the full action set $\mathbb{A}$ as the

union of the choice player action set $\mathbb{A}_\pi$, the chance player action set $\mathbb{A}_\psi$, and the terminal player action set $\mathbb{A}_d$ as follows: $\mathbb{A} = \mathbb{A}_\pi \cup \mathbb{A}_\psi \cup \mathbb{A}_d$. Let us introduce the special **no-op action**, which is produced when it is not a player's turn to move, such that $\forall n \in \mathbb{N}$ and $\forall s \in \mathbb{S}, \mathbb{A}_n(s) = \{\text{no-op}\}$ when $\iota(s) \neq n$. The action set $\mathbb{A}_d$ of the terminal player contains the no-op action as well as the **end action**, which represents ending the game: $\mathbb{A}_d = \{\text{no-op}, \text{end}\}$. We can see that for any given state, the set of legal chance actions and the set of legal choice actions are both subsets of the set of all actions: $\forall s \in \mathbb{S}, \mathbb{A}_\psi(s) \subseteq \mathbb{A}$, and $\forall s \in \mathbb{S}, \mathbb{A}_\pi(s) \subseteq \mathbb{A}$.

To provide input s^{k-1}, a^k to the dynamics function g_θ, the original MuZero encodes the action performed as an image and stacks it with the current hidden state. As with any choice player action, we may then encode any chance action $a \in \mathbb{A}_\psi$ taken from $\psi(s)$ as an image and stack it with a previous hidden state to produce input for g_θ.

For an agent to properly model a stochastic environment, it must determine when chance events occur as well as the distributions of their effects. The problem then is to learn an approximation of the player identity function $\iota(s)$ as well as the chance player policy $\psi(s)$. We then add two quantities that our model, conditioned by parameters θ, must predict: the **chance policy** $\boldsymbol{c}^k \approx \psi_{t+k}$ and the **player identity policy** $\boldsymbol{i}^k \approx \iota_{t+k}$ in addition to the quantities already present in MuZero: the policy $\boldsymbol{p}^k \approx \pi_{t+k}$ (which we now term the **choice policy**), value $v^k \approx z_{t+k}$, and reward $r^k \approx u_{t+k}$.

Two modifications are made to the MuZero network architecture. The prediction function now produces the choice policy, chance policy, and value: $\boldsymbol{p}^k, \boldsymbol{c}^k, v^k = f_\theta(s^k)$, while the dynamics function now produces the reward, next hidden state, and player identity policy: $r^k, s^k, \boldsymbol{i}^k = g_\theta(s^{k-1}, a)$, where the length of $\boldsymbol{i}^k = |\mathbb{N}|$. We use the no-op action as a training target for the choice policy and chance policy when it is not that player's turn to move.

Omitting the L2 regularization term, we define our loss function as follows:

$$l_t(\theta) = \sum_{k=0}^{K} l^r(u_{t+k}, r_t^k) + l^v(z_{t+k}, v_t^k) + l^p(\pi_{t+k}, \boldsymbol{p}_t^k) + l^c(\psi_{t+k}, \boldsymbol{c}_t^k) + l^i(\iota_{t+k}, \boldsymbol{i}_t^k) \tag{1}$$

3.3 Tree Search

Equipped with these new quantities, we wish to modify our MCTS to accommodate chance nodes. To flexibly capture the dynamics of stochastic games, the model must determine who is to play before either making a player action, making a chance action, or terminating the search. We introduce the **identity layer**, a layer of nodes that we interleave with **action layers**. The identity layer employs our ι approximation to determine the direction of the search.

Let us declare the root node to be a **choice node**. As in MuZero, the root hidden state s^0 is generated by passing observations into the representation function such that $s^0 = h_\theta((o_1, \ldots, o_t))$. We then derive the priors of the root node's children by feeding the root hidden state into the prediction function: $\boldsymbol{p}^0, _, _ = f_\theta(s^0)$; here, the chance policy and value may be ignored. Following MuZero, the children of the root node are taken from the set of legal actions derived from the true environment state, $\mathbb{A}_\pi(s^t)$, and assigned edges corresponding to those actions; likewise, we select among the children of a choice node with PUCT during tree search.

The children of choice nodes are **identity nodes**. Upon expansion, the reward r^k, next hidden state s^k, and identity policy $\boldsymbol{i}^k$ of an identity node is generated by applying the dynamics function to the hidden state of its parent s^{k-1} along with the image of its action: $r^k, s^k, \boldsymbol{i}^k = g_\theta(s^{k-1}, a)$. The prediction function is then applied to the next hidden state s^k, yielding the choice policy, chance policy, and value: $\boldsymbol{p}^k, \boldsymbol{c}^k, v^k = f_\theta(s^k)$. An identity node has a number of children equal to $|\mathbb{N}|$; thus, in our example each identity node has four children corresponding to $\{1, 2, c, d\}$. A child is selected from an identity node using roulette wheel selection proportional to $\boldsymbol{i}^k$. The children from the choice player edges $\{1, 2\}$ become choice nodes, the child from the chance edge c becomes a **chance node**, and the child from the terminal edge d becomes a **terminal node**.

Each **chance node** and nonroot choice node inherits the hidden state s^k from its parent identity node and has a number of children equal to $|\mathbb{A}|$. (We do not assume prior knowledge of $|\mathbb{A}_\pi|$ or $|\mathbb{A}_\psi|$; so long as we begin with $|\mathbb{A}|$, these are learned and approximated through training.) A choice node also inherits from its parent the value v^k, and its children take their priors from the choice policy $\boldsymbol{p}^k$. A chance node inherits the chance policy $\boldsymbol{c}^k$ from its parent and uses this to generate the priors of its

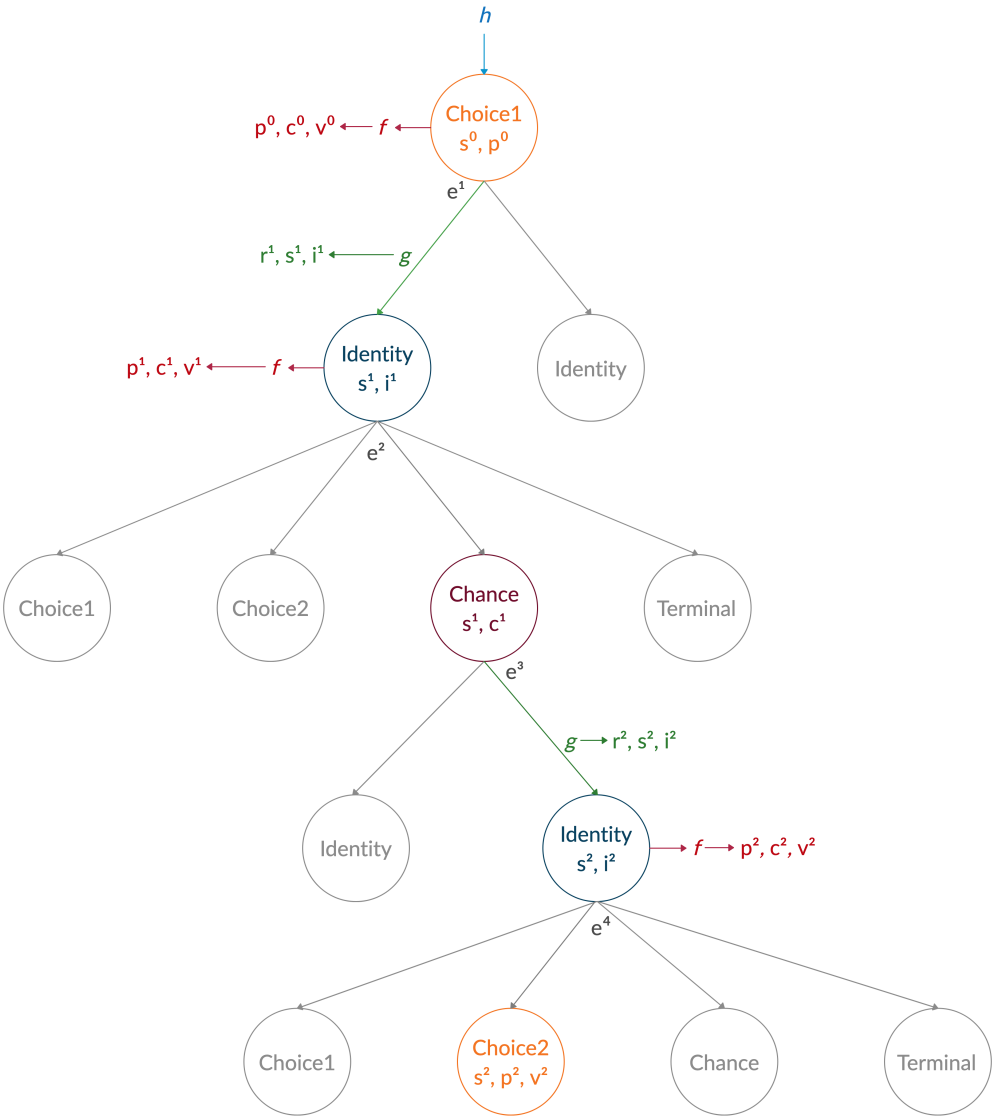


Figure 1: Illustration of NDMZ tree search. e^k represents the edge between nodes.

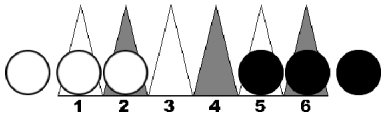
children; during search, the child of a chance node is also chosen using roulette wheel selection. All children of chance nodes and nonroot choice nodes are identity nodes.

In contrast with MuZero, backup is only performed when the leaf node is either a choice node or a terminal node. First, let us consider the case when the leaf node is a choice node. We will refer to identity nodes with a choice node parent as **result nodes**. We only employ PUCT during search when selecting among result nodes and use roulette wheel selection otherwise. Because of this, we only need add or subtract the value obtained from the leaf node to the result nodes found in the visit path. If the edge leading to the leaf node is the same as the edge leading to the parent of the result node, we add the value of the leaf node to the total value of the result node; otherwise, we subtract. If the leaf node is a terminal node, we start with the parent of the leaf node and search up the tree until we find the most recent choice node. We note the value and the edge leading to this most recent choice node, return to the leaf node, and proceed with backup as described above.

Pseudocode for a version of the algorithm applicable to board games lacking immediate rewards is given in full in Appendix A.

4 EXPERIMENTS

4.1 NANNON



For testing, benchmarking, and evaluation, we used Nannon, a simplified variant of backgammon that is solvable through value iteration (Pollack, 2005). Nannon is properly considered as a parameterized set of games, such that the number of board points, checkers per player, and sides of dice may be changed, influencing the complexity of the game. It is played by rolling a single die. There is no stacking, and only one checker per point is allowed; however, two adjacent pieces of the same color form a block. The number of possible board states may be found with the following equation, where n is the number of points on the board and k is the number of checkers per player:

$$\sum_{i=0}^{k}\sum_{j=0}^{k}\binom{n}{i}\binom{n-i}{j}(k+1-i)(k+1-j) \tag{2}$$

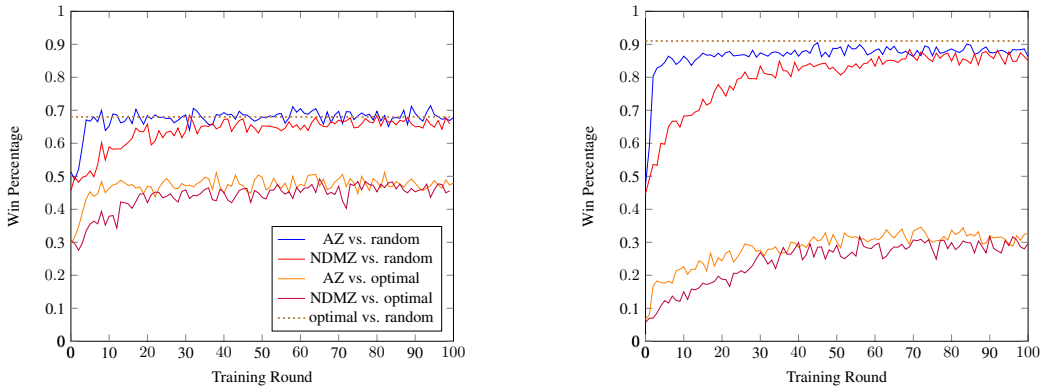
We chose two game configurations, both with a six-sided die: six points on the board with three checkers per player (6-3-6) and twelve points on the board with five checkers per player (12-5-6). 6-3-6 Nannon has 2,530 states, and 12-5-6 Nannon has 1,780,776 states. The first game configuration has some surprising subtleties despite the small number of total states; the second, while much more complex, is still far more tractable than backgammon. Each configuration was solved using value iteration, yielding a lookup table with which the optimal policy could be derived (Bellman, 1957). This policy is used to guide the actions of the **optimal player**; by pitting agents produced by AlphaZero and NDMZ against the optimal player, we obtain a more objective estimate of their performance.

4.2 EXECUTION AND HYPERPARAMETERS

The NDMZ algorithm was executed as follows. The Nannon game engine code was written in C++ using the OpenSpiel framework, while the algorithm was written in Python using Tensorflow 2.4. Dense multi-layer perceptrons were used instead of convolutional resnets. Models for f_θ, g_θ, and h_θ were initialized with two hidden layers each of 256 weights, using SELU activation and LeCun Normal initialization (Klambauer et al., 2017). Experiments were carried out on a Kubernetes cluster with Ray using 500 CPU workers for evaluation and 300 CPU workers for self-play; a single T4 GPU was used for training.

One hundred rounds of self-play, training, and evaluation were performed. For both AlphaZero and NDMZ, the models run for three hundred games of self-play per round, while the tree search performs one hundred simulations per move. The 6-3-6 run used a replay buffer limit of 4,000 games, while the 12-5-6 run used a limit of 1,000, as games on the larger board are roughly four times longer; for both configurations, the replay buffer carries about 100,000 examples total. For MuZero, a K value of six is used, and samples are pulled at random from randomly chosen games before being unrolled K times and trained via backpropagation through time. Around 600,000 examples are then seen per epoch for NDMZ and 100,000 for AlphaZero.

Training for both algorithms employs an Adam optimizer with a learning rate of 0.001 and a batch size of 512. We did not scale the hidden states to the same range as the action input, nor did we scale the loss by $\frac{1}{K}$ or $\frac{1}{2}$. For the first twenty rounds of the trial, five epochs of training are performed; afterwards, only a single epoch is done. Evaluation of the AlphaZero and NDMZ agents is performed once before training begins and after every training iteration, consisting of 1,000 rounds of play versus the optimal player and 1,000 rounds of play versus an agent that makes moves at random (500 games of each color).



(a) 6-3-6 Nannon Performance

(b) 12-5-6 Nannon Performance

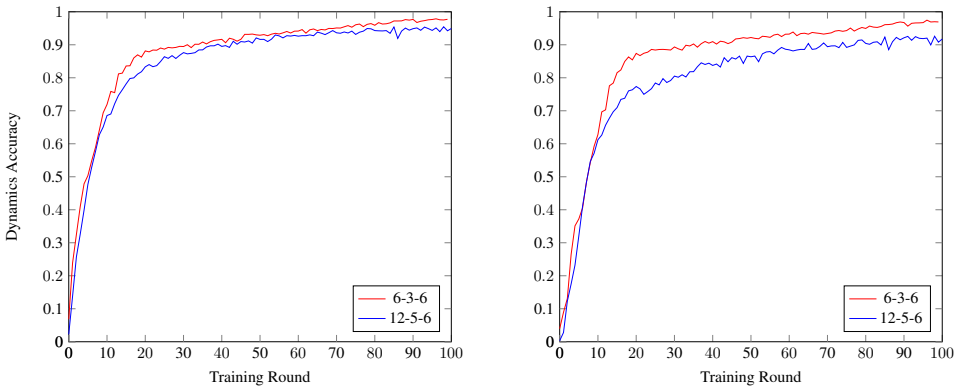
Figure 2: Performance evaluation results of agents produced by the NDMZ and AlphaZero (AZ) algorithms in the game of Nannon throughout the duration of a training run.

4.3 Dynamics Evaluation

Tests were designed to provide more practical insight into the ability of the networks to capture the dynamics of the game. After each training iteration, five hundred games are played with random moves and two tests are performed.

For the first test, the game trajectories are analyzed in the following manner: For every position at timestep t reached in every game, the choice policy output $\boldsymbol{p}^t$ of f_θ is compared with the actual legal moves for that position: $\mathbb{A}_\pi(s^t_{\text{env}})$. If the choice policy ranks any illegal move with greater probability than a legal move, the position is marked a failure; otherwise, it is a success. This proceeds recurrently for K times, using the dynamics network g_θ along with the move actually played to produce a new hidden state that is fed into f_θ, obtaining $\boldsymbol{p}^{t+k}$ for each k in $1, \ldots, K$. The actual legal moves for these future states $s^{t+1}, \ldots, s^{t+K}$ are compared with the recurrently produced policies, and the results for all positions are averaged. We call this the **top move dynamics test**.

The second test proceeds along similar lines as the first; however, the position is marked a failure if any illegal move is given greater than uniform probability. For example, if $\boldsymbol{p}^t$ has length 4 and an illegal move has probability > 0.25, the test for that position is a failure; otherwise it is a success. This we call the **uniform dynamics test**. Each test evaluates around 60,000 positions for 6-3-6 Nannon and 240,000 positions for 12-5-6 Nannon.



(a) Top Move Dynamics Test Results

(b) Uniform Dynamics Test Results

Figure 3: Results of dynamics tests as described in Section 4.3.

4.4 RESULTS AND CONCLUSION

We observe in Figure 2 that, for both configurations, AlphaZero experiences a very sharp and immediate increase in performance against the random player; the performance of NDMZ, on the other hand, exhibits a gradual curve before reaching a plateau of near-convergence with the AlphaZero agent. The dynamics experiments in Figure 3 show that NDMZ agents can predict valid moves with a high degree of accuracy even several moves deep in the search. By comparing the two graphs, we also observe that the performance of the agent improves as it better learns to model its environment.

From these results, we may conclude that NDMZ is capable of learning effective strategies and a useful model of the game. Future work could entail testing NDMZ on games with more unorthodox chance node interleavings, such as Chinese Dark Chess, and extending the algorithm to accommodate densely stochastic games.

REFERENCES

Bruce W. Ballard. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(3):327 – 350, 1983. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(83)80015-0. URL http://www.sciencedirect.com/science/article/pii/S0004370283800150.

Richard Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957. ISSN 0022-2518.

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

Guillaume Chaslot, Mark Winands, H. Herik, Jos Uiterwijk, and Bruno Bouzy. (DPW) progressive strategies for monte-carlo tree search. 04:343–357. doi: 10.1142/S1793005708001094.

Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *In: Proceedings Computers and Games 2006*. Springer-Verlag, 2006.

C. Hsueh, I. Wu, J. Chen, and T. Hsu. Alphazero for a non-deterministic game. In *2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pp. 116–121, 2018.

G. Klambauer, Thomas Unterthiner, Andreas Mayr, and S. Hochreiter. Self-normalizing neural networks. *ArXiv*, abs/1706.02515, 2017.

Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293 – 326, 1975. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(75)90019-3. URL http://www.sciencedirect.com/science/article/pii/0004370275900193.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. volume 2006, pp. 282–293, 09 2006. doi: 10.1007/11871842_29.

Marc Lanctot, Abdallah Saffidine, Joel Veness, Christopher Archibald, and Mark Winands. Monte carlo *-minimax search. 08 2013.

Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. Openspiel: A framework for reinforcement learning in games, 2019.

M. Maschler, E. Solan, S. Zamir, Z. Hellman, and M. Borns. *Game theory*. Cambridge University Press, Cambridge, 2013.

D. Mitchie. Game-playing and game-learning automata. In L. Fox (ed.), *Advances in Programming and Non-Numerical Computation*, volume 3 of *Proceedings of Summer Schools*, pp. 183–200. Pergamon, 1966.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015. doi: 10.1038/nature14236.

J. Pollack and A. Blair. Why did td-gammon work? In *NIPS*, 1996.

Jordan B. Pollack. Nannon: A nano backgammon for machine learning research. In *CIG*, 2005.

Christopher D. Rosin. (PUCT) multi-armed bandits with episode context. 61(3):203–230, 2011. ISSN 1573-7470. doi: 10.1007/s10472-011-9258-6. URL https://doi.org/10.1007/s10472-011-9258-6.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model, 2019.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. (AlphaZero chess) mastering chess and shogi by self-play with a general reinforcement learning algorithm. a. URL http://arxiv.org/abs/1712.01815.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. (AlphaZero go) mastering the game of go without human knowledge. 550(7676):354–359, b. ISSN 0028-0836, 1476-4687. doi: 10.1038/nature24270. URL http://www.nature.com/articles/nature24270.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016. ISSN 1476-4687. doi: 10.1038/nature16961. URL https://doi.org/10.1038/nature16961.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

G. Tesauro. Temporal difference learning and td-gammon. *J. Int. Comput. Games Assoc.*, 18:88, 1995.

François Van Lishout, Guillaume Chaslot, and Jos WHM Uiterwijk. Monte carlo tree search in backgammon. 2007. [Online]. Available: http://hdl.handle.net/2268/28469 (current 2007).

Shi-Jim Yen, Cheng-Wei Chou, Jr-Chang Chen, I-Chen Wu, and Kuo-Yuan Kao. Design and implementation of chinese dark chess programs. *IEEE Transactions on Computational Intelligence and AI in Games*, 7:1–1, 01 2014. doi: 10.1109/TCIAIG.2014.2329034.

S. Çalışır and M. K. Pehlivanoğlu. Model-free reinforcement learning algorithms: A survey. In *2019 27th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, 2019.

A APPENDIX

Algorithm Nondeterministic MuZero Tree Search

function NDMZSEARCH(s^{env}) ▷ s^{env} is the raw environment state
  create root node w_0
  EXPANDROOT(w_0, s^{env}) ▷ adds children and hidden state to root
  while within computational budget **do**
    $w_L \leftarrow$ TREESELECT(w_0) ▷ returns a leaf node
    if $K(w_L) =$ choice or $K(w_L) =$ terminal **then** ▷ K is the kind of node
      BACKUP(w_L, v)
  $\mathbb{W} \leftarrow$ children of w_0 ▷ N is the number of visits
  $w_1 \leftarrow$ sample a child w' from $\mathbb{W}$ proportional to $\frac{N(w')^{1/\tau}}{\sum_{b \in \mathbb{W}} N(b)^{1/\tau}}$ ▷ τ: temperature param
  return $A(w_1)$

function EXPANDROOT(w_0, s^{env})
  $S(w_0) \leftarrow h_\theta(s^{\text{env}})$ ▷ representation network assigns hidden state S to root.
  $K(w_0) \leftarrow$ choice ▷ root is choice node
  $\boldsymbol{p}, _, _ \leftarrow f_\theta(S(w_0))$ ▷ prediction network: choice policy logits
  $\boldsymbol{p} \leftarrow$ masked softmax of $(\boldsymbol{p}, \mathbb{A}_\pi(s^{\text{env}}))$ ▷ scale for legal actions only
  for $e \in \mathbb{A}_\pi(s^{\text{env}})$ **do** ▷ for each edge e taken from the set of legal actions
    add child w' with ▷ N: visit count; W: total value;
    $N(w') \leftarrow 0; W(w') \leftarrow 0; Q(w') \leftarrow 0;$ ▷ Q: mean value; P: prior probability;
    and $P(w') \leftarrow \boldsymbol{p}_a; E(w') \leftarrow e$ ▷ E: edge leading to node

function BACKUP(w)
  if $K(w) =$ terminal **then** ▷ if w is a terminal node
    $n \leftarrow w$
    while $K(n) \neq$ choice **do**
      $n \leftarrow$ parent of n ▷ search up the tree until we find the closest choice node
    $e \leftarrow E(n)$ ▷ player identity of the choice node
    $v \leftarrow V(n)$ ▷ value from the choice node
  else
    $e \leftarrow E(w)$ ▷ player identity of the leaf node
    $v \leftarrow V(w)$ ▷ value from leaf node
  while w is not none **do**
    $j \leftarrow$ parent of w
    if $K(w) =$ identity and $K(j) =$ choice **then** ▷ if w is a result node
      if $E(j) = e$ **then** ▷ if the edge leading to j is the same
        $W(w) \leftarrow W(w) + v$ ▷ add to total value
      else
        $W(w) \leftarrow W(w) - v$ ▷ subtract from total value
      $N(w) \leftarrow N(w) + 1$ ▷ increment visit count
      $Q(w) \leftarrow \frac{W(w)}{N(w)}$ ▷ adjust mean value
    $w \leftarrow$ parent of w

```
function TreeSelect(w)
    while K(w) is not terminal do
        if w not expanded then                                  ▷ w is a leaf node
            return Expand(w)                                    ▷ adds child nodes, hidden state, and value
        W ← children of w
        if K(w) = chance or K(w) = identity then                ▷ chance or identity node
            w ← sample w' from W weighted by P(w')              ▷ roulette wheel selection
        else if K(w) = choice then                              ▷ choice node
            w ← argmax_{w' ∈ W} ( Q(w') + c_puct P(w') √(Σ_{b∈W} N(b)) / (1 + N(w')) )   ▷ PUCT
    return w

function Expand(w)
    j ← parent of w
    if K(j) = choice or K(j) = chance then                      ▷ if w is the child of a choice or chance node
        K(w) ← identity                                         ▷ w is an identity node
        _, S(w), p ← g_θ(S(j), A(w))                            ▷ for board games, we ignore the reward
        O(w), C(w), V(w) ← f_θ(S(w))                            ▷ choice policy, chance policy, and value
        C ← N                                                   ▷ C is the set of child edges
    else if K(j) = identity then                                ▷ if w is the child of an identity node
        S(w) ← S(j)                                             ▷ w always inherits the hidden state of its parent
        if A(w) = c then                                        ▷ if the edge leading to w is the chance edge c
            K(w) ← chance                                       ▷ w is a chance node
            p ← C(j)                                            ▷ p is the chance policy vector
            C ← A                                               ▷ A is the full action set
        else if A(w) = d then                                   ▷ if the edge leading to w is the terminal edge d
            K(w) ← terminal                                     ▷ w is a terminal node
            C ← {}                                              ▷ terminal nodes have no children
        else
            K(w) ← choice                                       ▷ w is a choice node
            p ← O(j)                                            ▷ p is the choice policy vector
            V(w) ← V(j)                                         ▷ w inherits value from parent
            C ← A
    for e ∈ C do                                                ▷ for each edge in the set of child edges
        add child w' with
        N(w') ← 0; W(w') ← 0; Q(w') ← 0;
        and P(w') ← p_e; E(w') ← e                              ▷ assign prior to child from p along with edge e
    return w
```