Train on Small, Play the Large: Scaling Up Board Games with AlphaZero and GNN

Shai Ben-Assayag
Train on Small, Play the Large: Scaling Up Board Games with AlphaZero and GNN

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Shai Ben-Assayag

Submitted to the Senate of the Technion — Israel Institute of Technology
Av 5781 Haifa July 2021
This research was carried out under the supervision of Prof. Ran El-Yaniv, in the Faculty of Computer Science.

Some results in this thesis have been submitted as articles by the author and research collaborators in conferences and journals during the course of the author’s research period, the most up-to-date versions of which being:

<table>
<thead>
<tr>
<th>Title</th>
<th>Authors</th>
<th>Submission Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train on small, play the large: Scaling up board games with alphazero and gnn</td>
<td>Shai Ben-Assayag and Ran El-Yaniv</td>
<td>Submitted, 2021</td>
</tr>
</tbody>
</table>

The generous financial help of the Technion is gratefully acknowledged.
# Contents

## List of Figures

## Abstract

1 Introduction

2 Related Work

2.1 AI for Board Games

2.1.1 RL for Board Games

2.2 Graph Neural Networks

2.3 Scalable Deep Reinforcement Learning

3 Preliminaries

3.1 Reinforcement Learning

3.1.1 Introduction

3.1.2 Markov Decision Processes

3.1.3 Planning

3.2 Board Games RL Environment

3.3 AlphaZero

3.3.1 Introduction

3.3.2 Training Pipeline:

3.3.3 Monte Carlo Tree Search:

3.4 Graph Neural Networks

3.4.1 Introduction

3.4.2 The Message Passing Procedure

3.4.3 Graph Isomorphism Networks

3.4.4 Graph Convolutional Networks

4 The Scalable AlphaZero Model

4.1 Replacing the CNN

4.2 Guiding MCTS

4.3 Training Pipeline
5 Experiments and Results

5.1 Experimental Setup .................................................. 29
  5.1.1 Board Games .................................................. 29
  5.1.2 Training Setup .................................................. 30
  5.1.3 Opponents ....................................................... 31
  5.1.4 Measure of Success ............................................. 31

5.2 Model Analysis .......................................................... 32
  5.2.1 The Merits of our Modified Components: ....................... 32
  5.2.2 Generalization to Larger Boards ................................ 33
  5.2.3 Training Time Analysis ......................................... 34
  5.2.4 Comparison to AZ ................................................. 34
  5.2.5 Go Evaluation .................................................. 37

6 Conclusion and Future Directions ...................................... 39

A Appendix ........................................................................ 41
  A.1 Pseudocode ............................................................. 41
  A.2 Global Setup ........................................................... 41
  A.3 Gomoku Generalization Test ......................................... 45

Hebrew Abstract ................................................................  i
List of Figures

3.1 The agent–environment interaction loop in an RL framework. At each time step $t$: (1) the agent observes the state (any information available to the agent) and chooses an appropriate action (2) the environment “responds” by evolving through (probably) unknown dynamics and (3) the agent observes the new state and the immediate reward. 12

3.2 Relationships between experience, model, values, and policy in an RL model that uses planning. 14

3.3 At each game state $s$, visited during the selfplay stage, the player chooses an action according to the policy vector $\pi_s$ provided by the MCTS. 17

3.4 For each training example $s$ visited during the selfplay stage, the neural network: (1) minimizes the difference between $z$, the final result of the specific game, and $v_s$, the state value; and (2) maximizes the similarity between the MCTS policy vector $\pi_s$ and the neural network policy vector $p_s$. 17

3.5 At each round of an MCTS, the selection, expansion and backpropagation steps are repeated in order to produce an improved policy $\pi_{s_0}$. 19

3.6 After some rounds of the selection, expansion and backpropagation steps, the MCTS policy vector is set to be proportional to the action visit count. 19

3.7 At each round of *message passing*, every node gets messages from all its neighbours on the graph, aggregates them and updates its hidden representation accordingly. 21

4.1 Illustration of the graph $G(s_0)$ generated based on the initial board state, $s_0$, of the game Othello. The blue and green nodes correspond to the light and dark pieces, respectively. Our additional dummy node is the central red with black border node connected to all other nodes. It is the only node that does not represent any square on the board. 24

4.2 Neural network architecture. 25

4.3 Six random subgraphs of the graph presented in Figure 4.1b. The subgraphs are sent in batches to $f_{\theta}$, and then the average probability for playing each action is used to guide the MCTS procedure. 26
5.1 (a) An $8 \times 8$ Othello board. By placing a light stone at the star-marked square, all dashed lines would be colored in white. (b) A Gomoku board. The game’s objective is to place 5 consecutive pieces of the same color either in a row, column or diagonal. (c) A $19 \times 19$ Go board. In the area marked with a red square we can see that the player with the dark pieces occupied the entire territory by surrounding the light pieces from every optional direction.

5.2 Average outcome of scalable players against the reference opponents on various board sizes and games. The shadowed areas represent the standard errors (5 independent runs).

5.3 (a) The synthetic Othello boards of increasing sizes we created. A similar board of the same form was created for all board sizes between $6 \times 6$ and $350 \times 350$. (b) The first two principal components of the embeddings provided by our GNN (X,Y plane) as a function of the board size (Z axis). Blue points refer to the embedding of the “good action” of placing a dark piece in the top-left corner. Red ones refer to the “bad action” of placing a dark piece in the bottom-right corner.

5.4 The first three principal components of the embeddings provided by our GNN. Each point represent one board size between $6 \times 6$ and $350 \times 350$. Blue points refer to the embedding of the “good action” of placing a dark piece in the top-left corner. Red ones refer to the “bad action” of placing a dark piece in the bottom-right corner.

5.5 Progression of GNN skill along training. The average outcome is evaluated by playing against the greedy opponent on Othello (board size $16 \times 16$) and Gomoku (board size $17 \times 17$).
Abstract

Playing board games is considered a major challenge for both humans and AI researchers. Because some complicated board games are quite hard to learn, humans usually begin with playing on smaller boards and incrementally advance to master larger board strategies. Most neural network frameworks that are currently tasked with playing board games neither perform such incremental learning nor possess capabilities to automatically scale up.

In this work, we look at the board as a graph and combine a graph neural network architecture inside the AlphaZero framework, along with some other innovative improvements. Our ScalableAlphaZero is capable of learning to play incrementally on small-scale boards, and advancing to play on large ones. Our model can be trained quickly to play different challenging board games on multiple board sizes, without using any domain knowledge. We present an extensive empirical study in which we apply our model to three different board games, including the highly complex game of Go. We demonstrate the generalization power and the effectiveness of our model and show, for example, that by training it for only three days on small Othello boards, it can defeat the AlphaZero model on a large board, which was trained to play the large board for 30 days.
Chapter 1

Introduction

In recent decades, researchers have been fascinated by the task of playing board games. Learning to play board games that have vast state space and large branching factors often requires extremely complex tactics. Despite being solved completely for small-scale boards (e.g., by exhaustive search), when dealing with computers, this is still considered highly challenging for large ones. The study of generalization in AI board game playing was motivated by this observation.

Learning a simple instance of a problem with the goal of solving a more complicated one is a common approach within various fields. Both humans and AI programs use such incremental learning, particularly when the large-scale problem instance is too hard to learn from scratch or too expensive. This thesis is concerned with applying incremental learning to the challenge of mastering board games. When playing board games, humans have the advantage of being able to learn the game on a small board, recognize the main patterns, and then implement the strategies they have acquired, possibly with some adjustments, on a larger board. In contrast, machine learning (ML) algorithms usually cannot generalize well between board sizes. While simple heuristics, such as zero padding of the board or analyzing local neighborhoods, can alleviate this generalization problem, they do not scale well for enlarged boards (see, e.g., Chapter 5).

Reinforcement learning (RL) is a branch of ML that is widely used for learning to play board games. In its basic form, an RL agent aims to deal with sequential decision making based on interactions with the environment. Each time a decision is made, the agent observes the new environment state and receives a reward signal. Using a trial and error process, the agent aims to find a policy that maximizes his accumulated rewards. In board games, the decisions refer to the actions on the board, and the rewards are usually determined by the final result of the game (e.g., a win, a tie or a loss).

In this thesis we propose ScalableAlphaZero (SAZ), a deep RL based model that can generalize to multiple board sizes of a specific game. SAZ is trained on small boards and is expected to scale successfully to larger ones. Our technique should be usable for
scalable board games, whose rules for one board size apply to all feasible board sizes (typically, infinitely many). For instance, Go is scalable but standard chess is not.

A strong motivation for finding such a model is a potential substantial reduction in training time. As we demonstrate in this thesis, training a model on small boards takes an order of magnitude less time than on large ones. The reason is that the dimension of the states as well as the space of training examples are significantly smaller. This small-scaled learning approach also enables the system to operate entirely using modest computational resources. Moreover, training time is reduced because gameplay takes fewer turns to complete. This last observation could improve the quality of training from a different perspective, as it reduces the number of actions required before a reward (that is, the game result) is observed. Another motivation for creating a scalable player is its versatility. The player can play any desired board size simply by training one model. For example, when playing against a scalable opponent in an online gaming interface, users can select their preferred board size based on their skill level.

The proposed model is based on two modifications of the well-known AlphaZero (AZ) algorithm [SHS+17]. To the best of our knowledge, presently AZ is the strongest superhuman RL based system for two-player zero-sum games. AZ combines a neural network architecture and a Monte Carlo tree search (MCTS) [Cou06, CBSS08] and is trained by selfplay iterations. The main drawback of AZ is that it limits the user to training and playing only on a specific board size. This is the result of using a convolutional neural network (CNN) [AHM87] for predictive pruning of the AZ tree search. To overcome this obstacle, in SAZ we replace the CNN by a graph neural network (GNN) [SGT+08]. The GNN is a scalable neural network, i.e., it is an architecture that is not tied to a fixed input dimension. GNN’s scalability enables us to train and play on different board sizes and allows us to scale up to arbitrarily large boards with a constant number of parameters. To further improve the AZ tree search pruning, we propose an ensemble-like node prediction using subgraph sampling; namely, we utilize the same GNN for evaluating a few subgraphs of the full board and then combine their scores to reduce the overall prediction uncertainty.

We conduct experiments on three scalable board games and measure the quality of SAZ by comparing it to various opponents on different board sizes. Our results indicate that SAZ, trained on a maximal board size of $9 \times 9$, can generalize well to larger boards (e.g., $20 \times 20$). Furthermore, we evaluate it by competing against the original AZ player, trained on a large board. Our model, with around ten times less training (computation) time on the same hardware, and without training at all on the actual board size that was used for playing, performs surprisingly well and achieves comparable results.

The main contributions of this work are:

1. A model that is capable of successfully scaling up board game strategies. As far as we know this is the first work that combines RL with GNNs for this specific
2. A subgraph sampling technique that effectively decreases prediction uncertainty of GNNs in our context and is of potential independent interest.

3. The presentation of extensive experiments, demonstrated on three different board games, showing that our model requires an order of magnitude less training time than the original AZ but, still, can defeat AZ on large boards.

This thesis is organized as follows: We survey related works in Chapter 2. Chapter 3 includes relevant scientific background for this work. In Chapter 4, we describe our SAZ model in detail. In Chapter 5 we describe our experiments and discuss the results from different perspectives. Lastly, in Chapter 6 we conclude and discuss future directions for this research.
Chapter 2

Related Work

The solution proposed in this thesis instantiates a GNN model inside the AlphaZero model for the task of scalable board game playing. In this section, we review early work in AI and board games, focusing on the AlphaZero [SHS+17] algorithm. We further describe the GNN design and review various works that use GNN to guide an RL model. Finally, we summarize existing methods that aim to deal with scalable board games and accelerate the generalization between board sizes.

2.1 AI for Board Games

Historically, the majority of AI programs designed for playing board games utilized methods of tree search, including brute force mechanisms and pruning techniques. The minimax algorithm, which aims at minimizing the loss for the worst case scenario, was used for Chess playing in [Tur53]. Minimax variations, such as the alpha-beta [KM75] pruning algorithm, were widely used for game playing over the years, and were a key to creating the world’s strongest players in various games. Schaeffer introduced the first computer program to achieve superhuman skills on a complex board game (which had not been solved until then) [SCT+92]. The program used deep search along with some complicated evaluation functions and endgame databases, and showed remarkable results on the Checkers game; however, it was still defeated by the Checkers world champion. Two years later, an improved program, which is described in [SLLB96], could beat the Checkers world champion.

The Deep Blue program, presented in [SP97, CHJH02], is also noteworthy. Deep Blue is considered the first computer program to demonstrate superhuman Chess skills. This system used a grand-master game database, a Chess specific alpha-beta search and complex evaluation functions to beat the Chess world champion, Garry Kasparov. Another Chess computer program, Stockfish, was named the top computer Chess engine in 2015, holding the title until 2018. Stockfish uses alpha-beta search and complex domain-specific adaptations to ensure optimal results.
2.1.1 RL for Board Games

Given an optimization problem, deep RL aims at learning a strategy for maximizing the problem’s objective function. The majority of RL programs do not make use of expert knowledge about the environment. Instead, they learn the optimum strategy by exploring state and action spaces with the goal of maximizing their cumulative rewards. For further details about RL, see Section 3.1.

Samuel introduced the first work that taught itself to play a board game (Checkers) by playing against itself, and that is very similar to what we call RL now [Sam59]. Tesauro presented the top-human backgammon player TD-gammon, which employs a temporal difference (TD-\(\lambda\)) RL algorithm and neural network training [TG96]. The program was trained via selfplay (a few million episodes) and rollouts. Rollout is the process of estimating the value of a state \(s\) by playing the game many times, starting from \(s\), with random dice rolls. Further reading on rollouts may be found at Subsection 3.1.3.

AlphaGo (AG) [SHM+16] is an RL framework that employs a policy network trained with examples taken from human games, a value network trained by selfplay, and MCTS, which defeated a professional Go player in 2016. About a year later, AlphaGo Zero (AGZ) [SSS+17] was released, improving AlphaGo’s performance with no handcrafted game specific heuristics; however, it was still tested only on the game of Go. AlphaZero [SHS+17] validated the general framework of AGZ by adapting the same mechanism to the games of Chess and Shogi. AG and AGZ have a three-stage training pipeline: selfplay, optimization and evaluation, whereas AZ skips the evaluation step. AGZ and AZ do not use their neural network to make move decisions directly. Instead, they use it to identify the most promising actions for the search to explore, as well as to estimate the values of nonterminal states. Section 3.3 provides full description of the AZ model, including the training procedure and the MCTS algorithm.

2.2 Graph Neural Networks

GNNs, introduced in [SGT+08], are a promising family of neural networks for graph structured data. GNNs have shown encouraging results in various fields including natural language processing, computer vision, logical reasoning and combinatorial optimization. Over the last few years, several variants of GNNs have been developed (e.g., [HYL17, GSR+17, LTBZ15, VCC+17, DBV16]), while the selection of the actual variant that suits the specific problem depends on the particularities of the task.

In their basic form, GNNs update the features associated with some elements of an input graph denoted by \(G = (V, E)\), based on the connections between these elements in the graph. A message passing algorithm iteratively propagates information between nodes, updates their state accordingly, and uses the final state of a node, also called “node embedding”, to compute the desired output. Section 3.4 provides more details...
about GNNs and the message passing procedure. In this thesis we use graph isomorphism networks (GINs) [XHLJ18] and graph convolutional networks (GCNs) [DBV16], which are a powerful well-known variants of GNNs. For further details about GINs and GCNs, see Subsection 3.4.3 and Subsection 3.4.4, correspondingly.

2.3 Scalable Deep Reinforcement Learning

Recently, several works tackled the problem of scalability in RL in the context of combinatorial optimization using GNNs that are natural models to deal with such challenges. For example, Lederman utilized the REINFORCE algorithm [Wil92] for clause selection in a QBF solver using a GNN, and successfully solved arbitrary large formulas [LRLS18]. Abe combined Graph Isomorphism Networks [XHLJ18] and the AGZ framework for solving small instances of NP-complete combinatorial problems on graphs [AXSS19]. Dai proposed a framework that combines RL with structure2vec graph embedding [DDS16], to construct incremental solutions for Traveling Salesman and other problems [DKZ+17]. Other RL models that deal with combinatorial optimization problems include [YP19, XT20].

A fundamental difference between trying to scale combinatorial optimization problems and our task is that a reductionist approach is much less intuitive for scaling up board games. For example, when trying to solve a large-scale SAT instance (as in [YP19]), the problem necessarily gets smaller as long as the search advances. More specifically, by setting a literal \( x_i \) to True or False, all clauses that contain \( x_i \) or \( \neg x_i \) can be deleted (either in conjunctive normal form or disjunctive normal form). For instance, setting \( x_1 \) to True in the following formula yields:

\[
(x_1 \lor x_4 \lor \neg x_5) \land (\neg x_1 \lor x_2 \lor \neg x_3) \land (x_4 \lor x_5 \lor \neg x_2) \xrightarrow{x_1 \leftarrow \text{true}} (x_2 \lor \neg x_3) \land (x_4 \lor x_5 \lor \neg x_2).
\]

In contrast, in a board game, the problem size remains the same during the entire search, with much more challenging rare boards that have not yet been encountered.

Among existing work on using learning to scale up board games, the most similar to our approach is that of [SS09]. To enable size generalization for Go-inspired board games, they presented MDLSTM, a scalable neural network based on MDRNNs and LSTM cells, computing four shared-weight swiping layers, one for each diagonal direction on the board. For each position on the board, they combine these four values into a single output representing the action probabilities. Their results show that MDLSTM transfers the strategies learned on small boards to large ones, leading to a level of play on a \( 15 \times 15 \) board that is on par with human beginners. Some other similar approaches include those of [GS10, WB07]. Gauci extrapolated \( 5 \times 5 \) Go solutions to \( 7 \times 7 \), thus speeding up the training [GS10]. Wu designed a DAG-RNN for Go and demonstrated that systems trained using a set of \( 9 \times 9 \) amateur games achieve surprisingly high correlation to the strategies obtained by a \( 19 \times 19 \) professional players’ test.
set [WB07].

All the above models, aimed at scaling up board games, do not incorporate an RL framework within their model, neither for training nor playing. In contrast to our model, which starts its training as a *tabula rasa* (i.e., without using any specific domain knowledge), the training processes of Schaul and Gauci are based on playing against a fixed heuristic based opponent, while Wu trained their model using records of games played by humans.
Chapter 3

Preliminaries

3.1 Reinforcement Learning

The following section provides an overview of RL. We follow [SB18], which provides an introduction and the basic notations of RL.

3.1.1 Introduction

RL is a branch of ML that focuses on interactions with the environment. The learner, also called the agent, makes sequential decisions with no prior knowledge about their outcome, and observes the changes in the environment after his actions are processed. The goal of the agent is to maximize its rewards. This can be seen as an abstraction of learning the cause and effect of every single action that can be taken and, in doing so, learning which actions should be taken to achieve one’s goals. RL is the paradigm of ML that is the most similar to a human (or animal) process of learning, in which learning happens after observing how the environment reacts. Figure 3.1 illustrates the agent-environment interaction in an RL framework.

Apart from the agent and the environment, the main elements of an RL system are:

1. **Policy**: A function that maps environment states to probabilities of selecting each action. It defines the behavior of the agent at any given time. The agent’s goal is to learn an optimal policy, that is a behavior that maximizes its goals. A policy is denoted by \( \pi(s) \) for a deterministic policy and \( \pi(a|s) \) for a stochastic policy.

2. **Reward signal**: At each time step the environment sends the agent a numerical value \( r \in \mathbb{R} \) representing the reward for his last action and the current state. The agent’s objective is to maximize the cumulative rewards over the entire interaction. It can be seen as the “good” and “bad” feedback that the learner receives from the environment. Agents influence rewards through their actions, either directly or indirectly by changing the environment. Future and sparse rewards
Figure 3.1: The agent–environment interaction loop in an RL framework. At each time step \( t \): (1) the agent observes the state (any information available to the agent) and chooses an appropriate action (2) the environment “responds” by evolving through (probably) unknown dynamics and (3) the agent observes the new state and the immediate reward.

3. **Value:** The value of a state is the total rewards that are expected to be given over the future, starting from that state. It takes into account the states that are more likely to be followed and their corresponding rewards. The intuition for estimating the value comes from the scenario in which the agent receives a low immediate reward, but later encounters a positive situation with a high reward. A high quality agent makes decisions based on the value functions rather than being distracted by immediate rewards.

4. **Model:** A function that attempts to learn the behavior of the environment. For example, predicting the next state given the current state and the action taken by the agent. Models are considered useful when the environment does not provide complete information or when calling the environment functions is computationally expensive. Methods that use models and planning (see Subsection 3.1.3) are called *model-based* methods, while trial-and-error learners are referred to as *model-free* methods.

RL differs from the well-studied supervised learning approach. With supervised learning, one learns from a training set of labeled examples, with the goal of generalizing to examples from the same distribution as the training set but not necessarily appearing in it. Interactive systems are not suitable for this type of learning, as such generalization to new situations is often difficult. An RL system typically develops its own labeled data through trial and error. For example, when trying to approximate a value function to match future rewards from the environment. However, in this case, the labels of the same state might change as the information about the environment grows. RL also differs from unsupervised learning, which identifies patterns in unlabeled data. In RL,
unlabeled data is used, but the goal is to find an optimal policy rather than to find hidden structure information.

### 3.1.2 Markov Decision Processes

The Markov decision process (MDP) describes the forms of an RL problem mathematically. The Markov property of MDP problems states that the response of the environment at time $t + 1$ depends only on the states and actions of time $t$ (and not on the entire history). We denote by $S$ the set of all possible states, by $A(S_t)$ the set of all possible actions available from the state $S_t$, and by $R \subseteq \mathbb{R}$ the set of possible numerical rewards. $S_t, A_t, R_t$ represent the state, action and reward at time step $t$. In this work, we consider the finite MDP case (i.e., $S, A$ are finite) with discrete time steps.

The environment’s one-step dynamics are defined by a function $p : S \times R \times S \times A \rightarrow [0, 1]$ where:

$$p(s', r, s, a) = \Pr[R_{t+1} = r, S_{t+1} = s' | S_t = s, A_t = a],$$

i.e., it is the conditional probability to reach the state $s'$ and receive a reward $r$, given the current state $s$ and the chosen action $a$. Based on the dynamics, we can derive other relevant information such as the expected reward,

$$r(s, a) = \mathbb{E}[R_{t+1} | S_t = s, A_t = a] = \sum_{r \in R} \sum_{s' \in S} p(s', r | s, a),$$

and the state-transition probabilities,

$$p(s' | s, a) = \Pr[S_{t+1} = s' | S_t = s, A_t = a] = \sum_{r \in R} p(s', r | s, a).$$

The return function is the amount of future rewards and is defined by:

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

where $t$ is a given time step, $T$ is the total number of interactions (or decisions) and $\gamma$ is the discount factor. $\gamma$ can be viewed as the agent’s certainty regarding future rewards. Recall that the agent’s goal is to maximize future rewards, which is mathematically equivalent to maximizing the expected return function. When $\gamma = 0$ only immediate rewards are maximized, and when $\gamma \to 1$, future rewards influence the agent’s decisions more. It is appropriate to apply a discount factor when an unlimited number of interactions are involved.

The MDP’s value function estimates the expected return of the agent given that he follows a given policy $\pi$ (also known as state-value function for the policy $\pi$) and is
formally defined by:

\[ v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s \right]. \]

Similarly, \( q_\pi(s, a) \) (also known as the action-value function for the policy \( \pi \)) encodes the expected rewards starting from \( s \), taking the action \( a \) and following the policy \( \pi \):

\[ q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a \right]. \]

Usually, \( v(\cdot) \) and \( q(\cdot, \cdot) \) are estimated by averaging the actual rewards that have followed the state. Such algorithms are called Monte Carlo methods.

### 3.1.3 Planning

Planning refer to deciding how to act, without necessarily experiencing the future. This situation involves the creation or improvement of a policy function using a model. At the core of this approach lies the future event look-ahead, the computation of the expected value by the model and, finally, the backpropagation of the value to all visited states. Given a state and a selected action, the model produces potential transitions and their probabilities of occurring. Figure 3.2 depicts an RL algorithm that is based on planning.

![Figure 3.2: Relationships between experience, model, values, and policy in an RL model that uses planning.](image)

In planning, it is common to focus on a certain state. In each new state \( S_t \), a planning process is performed, with the goal of selecting an optimal action \( A_t \). The
process continues when visiting $S_{t+1}$ and so on. This kind of planning is known as decision-time planning. It is often useful for tasks that do not require fast computation (such as in chess, where a player has several minutes to make a decision). In decision-time planning algorithms, state estimates are used immediately and then discarded, resulting in an efficient computation.

Rollout algorithms are special types of decision-time planning algorithms. The goal of rollout algorithms is to improve a given (and fixed) rollout policy, $\pi$, rather than to find an optimal one. In this method we follow the policy $\pi$ and simulate many trajectories that begin with a given state. An improved policy is updated by averaging the returns from all simulations. Once enough simulations have been performed, one of the actions with the highest estimated return is chosen and the process continues with the new state. Notice that the simulations are independent, thus enabling parallel computation. Several factors affect the time it takes to run a simulation: (1) the number of possible actions from each state; (2) the number of decisions to make before getting meaningful returns; and (3) the computational complexity of the rollout policy. The rollout policy and the accuracy of the value estimates have a large impact on the quality of the improved policy. Generally, more simulations result in better value estimates.

3.2 Board Games RL Environment

The board game environment encodes the rules of the game and maintains the board state. Players must obey these rules (i.e., make legal moves), and following their move, the game environment updates the state accordingly. We denote by $\mathcal{A}$ the set of possible actions and by $\mathcal{S}$ the set of possible board states. In addition, the environment provides a function $h : \mathcal{S} \to [-1, 1]$ that determines whether the game is ongoing or terminated with one of its possible results, usually a win (with reward 1), a tie (with reward 0) or a loss (with reward −1). As the reward is only provided when a final state is visited, this presents the agent with the challenge of sparse reward.

In a two-player board game the rival policy is not known in advance, so subsequent outcomes are uncertain. As a result, fixing the rival policy is necessary in order to frame this as an MDP. This can be accomplished by modifying the rival policy to be the same as the current policy (i.e., both sides use the same policy).

3.3 AlphaZero

In this section we provide relevant information about the AlphaZero algorithm as presented in [SHS+17].
3.3.1 Introduction

As we mentioned in Section 2.1, Silver proposed an RL algorithm for playing board games [SHS+17]. AZ is an improvement mechanism based on prior publications [SHM+16, SSS+17]. It provides a more general framework that can handle a variety of board games without any domain specific knowledge about the games except their rules. The system defeated Stockfish at Chess and all previous versions of AlphaGo at Go (note that AlphaGo itself defeated previous world champion Lee Sedol, ranked second in international titles).

AZ is a decision-time planning algorithm, based on a version of the rollout algorithm described in Subsection 3.1.3, which uses experience to guide its decisions. The neural network, which can be viewed as an intuition about the game’s states and actions, is combined with an MCTS, which is the core of the trial and error process. The neural network $f_\theta : \mathcal{S} \rightarrow [0, 1]^{|A|} \times [-1, 1]$ is used to guide the internal steps of the MCTS. $f_\theta$ gets as input a board state $s$ and outputs a probability vector for all possible moves $p_s$, and a scalar $v_s \in [-1, 1]$, which corresponds to the network’s confidence regarding the current player’s chances of winning the game. Subsection 3.3.2 describes the training pipeline and Subsection 3.3.3 describes in detail the MCTS procedure.

3.3.2 Training Pipeline:

The training is composed of a loop between two independent stages:

- **Selfplay:** The player plays against itself, using MCTS guided by the latest weights of $f_\theta$. The selfplay stage accumulates training examples of the form $(s, \pi_s, z_s)$, where $s \in \mathcal{S}$ is the state (usually in a canonical form), $\pi_s$ is the probability vector obtained from the MCTS and $z_s$ is the final result of the game (when using the canonical form for $s$, we always take the perspective of a specific player). At the end of this stage, AG and AGZ update the training set to include all the boards that can be constructed by a rotation or reflection of an example in the training set. Figure 3.3 illustrates the selfplay stage action choosing mechanism.

- **Optimization:** After constructing the training set in the previous stage, the neural network is trained to maximize the similarities between $p_s$ and $\pi_s$, and to minimize the difference between $v_s$ and $z_s$. The loss function used to achieve this goal (for a single example) is:

$$
\mathcal{L}(s) = (z_s - v_s)^2 + \text{CrossEntropy}(p_s, \pi_s) + c||\theta||_2^2
$$

where $c$ is a regularization factor. For an illustration of the neural network training procedure see Figure 3.4.

The training examples are kept between iterations. When one iteration ends, the oldest training examples are partially removed.
3.3.3 Monte Carlo Tree Search:

The tree search is designed to explore the game states and actions, and to provide an improved probability vector $\pi_s$. Here we describe the MCTS variant used in the AZ framework. For each pair $(s, a)$, corresponding to the state and action, it stores the following variables:

- $Q(s, a)$: The action value.

Figure 3.3: At each game state $s$, visited during the selfplay stage, the player chooses an action according to the policy vector $\pi_s$ provided by the MCTS.

Figure 3.4: For each training example $s$ visited during the selfplay stage, the neural network: (1) minimizes the difference between $z$, the final result of the specific game, and $v_s$, the state value; and (2) maximizes the similarity between the MCTS policy vector $\pi_s$ and the neural network policy vector $p_s$. 

Technion - Computer Science Department - M.Sc. Thesis MSC-2021-24 - 2021
• $P(s,a)$: The probability of choosing $a$ from the state $s$

• $N(s,a)$: The visit count of the pair $(s,a)$.

• $U(s,a)$: Upper confidence bound of the pair $(s,a)$ computed by:

$$U(s,a) = c_{\text{puct}} P(s,a) \sqrt{\frac{\sum_{a' \in \mathcal{A}} N(s,a')}{1 + N(s,a)}}$$

where $c_{\text{puct}}$ is a hyperparameter that controls the exploration and exploitation. When $c_{\text{puct}}$ is large the unvisited actions are more likely to be chosen. When $c_{\text{puct}}$ is small it is more likely to choose high-valued actions that have already been explored.

Each round of an MCTS starts with an initial state $s_0$ that we want to explore and evaluate, and consists of the following steps:

1. **Selection:** Start at the root $s_0$ and select a child node maximizing

$$Q(s,a) + U(s,a)$$

until an unexpanded node $s'$ is reached.

2. **Expansion:** If $s'$ is a terminal state (i.e., has a decisive result $z \in \{-1,0,1\}$, a win, a tie or a loss), let $v_{s'} = z$. Otherwise, evaluate $f_\theta(s') = (p_{s'}, v_{s'})$ and store $P(s', \cdot) = p_{s'}$

3. **Backpropagation:** Traverse all the pairs $(s,a)$ visited along the path to $s'$ and update:

$$Q(s,a) = \frac{N(s,a) \cdot Q(s,a) + v_{s'}}{N(s,a) + 1} \quad (3.1)$$

$$N(s,a) = N(s,a) + 1 \quad (3.2)$$

Figure 3.5 illustrates the selection, expansion and backpropagation steps.

For playing purposes, after a predefined number of repetitions, MCTS calculates the improved probability vector $\pi_{s_0}$. The vector element in the location corresponding to the action $a$ is defined as:

$$\pi^{a}_{s_0} = N(s,a)^{1/\tau}$$

where $\tau$ is a temperature parameter. When $\tau$ is large, the probability vector is much closer to a uniform distribution; when $\tau \to 0$, the probability of the most visited action is closer to 1. Usually, we reduce $\tau$ as the learning advances.

The rationale behind setting the policy vector proportional to the node visit count is that at the **selection** step, we always choose the action that maximizes $Q(s,a) + U(S,a)$.  

18
After traversing the tree enough times, we are likely to visit the most promising actions more frequently than the less promising ones (statistically). Figure 3.6 illustrates the playing mechanism of the MCTS player.

Figure 3.6: After some rounds of the selection, expansion and backpropagation steps, the MCTS policy vector is set to be proportional to the action visit count.

3.4 Graph Neural Networks

In this section we provide information about the GNN general framework and focus on the GNN variants that are relevant for this thesis.

3.4.1 Introduction

GNNs have recently become very popular among deep neural network researchers, showing promising results in a variety of fields. GNNs deal with graph structured data consisting of nodes, edges (relationships between nodes) and information about each
entity. Graph structured data offers a natural way to represent abstract concepts such as relationships and interactions. Notice that both images and text/time series can be represented in a graph form. An image can be modeled as a grid-like graph where nodes represent pixels and edges are defined by neighboring pixels. The text’s corresponding graph can be defined as a path graph, where the nodes represent words/characters, and the edges link adjacent words/characters. In conclusion, graph structured data is a more general method of storing data, but at the same time it can be much more difficult to interpret.

In a general GNN, a standard message passing architecture, described in Subsection 3.4.2, is applied for a number of rounds. Every node value is aggregated, passes through an activation function and then the node value is updated. The final node values, also known as node embeddings, are then passed on to the final layers that compute the output based on the specific task that has to be learned.

GNNs can be used for various use cases including: (1) graph classification [EPBM19, GSR+17]. For example, to identify the chemical reaction class of a molecular structure; (2) node classification/regression. For instance, to determine whether a Facebook profile is a bot or not; (3) edge prediction [ZC18, BKW17]. For example, to predict whether Facebook users know each other based on their friends’ connections; (4) clustering [BGA20, TPPM20, WPL+17]; (5) graph generation [LVD+18, DCK18, YLY+18]; and more.

### 3.4.2 The Message Passing Procedure

The *message passing* algorithm is a central component of graph neural networks. It uses a predefined number of iterations to propagate information between nodes on the graph. In its basic form, the message passing algorithm receives as input a graph \( G = (V,E) \) and the number of overall iterations \( T \), and stores hidden representations \( h^t_k \in d_t \) of the graph nodes, where \( k \in V, t \in \{1,\cdots,T\} \) and \( d_t \) is the hidden dimension of layer \( t \).

At iteration \( t \), each node \( k \) receives messages from its graph neighbors, denoted by \( N(k) \). Messages are generated by applying a message function \( m(\cdot) \) to the hidden states \( h^t_i \) of the nodes in the graph, and then are combined by an aggregation function \( \text{AGG}(\cdot) \), e.g., usually a sum or a mean (Equation 3.3). An update function \( u(\cdot) \) is later used to compute a new hidden state \( h^{t+1}_k \) for every node \( k \) (Equation 3.4). Figure 3.7 provides an overview of the message passing procedure on a toy graph. Finally, after \( T \) iterations, a readout function \( g(\cdot) \) outputs the final prediction, based on the final node embeddings \( h^T_k \) (see Equation 3.5 for node prediction output function and Equation 3.6 for graph prediction output function). Neural networks are often used for both \( m(\cdot) \), \( u(\cdot) \) and \( g(\cdot) \). Notice that in this case (for \( u \) and \( m \)), at a specific message passing iteration, the same neural network is applied to all the nodes of the graph (using the same weights). However, it is possible (and recommended) to use different networks...
for different iterations of message passing.

\[
M_{k}^{t+1} = \text{AGG} \sum_{i \in N(k)} m(h_{k}^{t}, h_{i}^{t}) \tag{3.3}
\]

\[
h_{k}^{t+1} = u(h_{k}^{t}, M_{k}^{t+1}) \tag{3.4}
\]

\[
o_{k} = g(h_{k}^{T}, h_{k}^{0}) \tag{3.5}
\]

\[
o_{G} = g(\text{AGG} \sum_{k \in V} h_{k}^{T}) \tag{3.6}
\]

Figure 3.7: At each round of message passing, every node gets messages from all its neighbours on the graph, aggregates them and updates its hidden representation accordingly.

### 3.4.3 Graph Isomorphism Networks

Xu proved that the graph isomorphism network (GIN) model is as powerful as the Weisfeiler-Lehman graph isomorphism test and is the most expressive model among the class of GNNs [XHLJ18]. We describe a hidden feature update layer of GIN, from a message passing perspective. During iteration number \(t\), each node \(k\) is updated as follows:

\[
h_{k}^{t+1} = h_{\theta} \left( (1 + \epsilon) h_{k}^{t} + \sum_{j \in N(k)} h_{j}^{t} \right)
\]

where node features are aggregated by a summation operation, \(\epsilon\) is either a learnable parameter or a fixed scalar and \(h_{\theta}\) denotes a neural network (i.e., an MLP). The same
update rule can be computed in a matrix form as:

\[ H^{t+1} = h_\theta \left( (A + (1 + \epsilon) \cdot I) \cdot H^t \right) \]

where \( A \) is the adjacency matrix of \( G \) and \( I \) is the identity matrix of the same dimensions. Note that in our GNN architecture we used a two headed network for computing the policy (node regression task) and the value (graph classification task).

3.4.4 Graph Convolutional Networks

The general idea behind graph convolutional neural networks (GCNs) [DBV16] is to apply convolution over a graph. We describe a hidden feature update layer of GCN, from a message passing perspective. At iteration number \( t \), each node \( k \) is updated by:

\[ h_{k}^{t+1} = \sum_{i \in \mathcal{N}(k) \cup \{k\}} \frac{1}{\sqrt{\deg(k) \cdot \deg(i)}} (\Theta^{t+1} \cdot h_{i}^{t}) \]

where neighboring node features are first transformed by a weight matrix \( \Theta \), normalized by their degree, and finally aggregated by a summation operation. The same update rule can be computed in a matrix form as:

\[ H^{t+1} = \hat{D}^{-1/2} \hat{A} \hat{D}^{-1/2} H^t \Theta^{t+1} \]

where \( \hat{A} = A + I \) is the adjacency matrix of \( G \), with inserted self loops, and \( \hat{D} \) is the corresponding diagonal degree matrix.
Chapter 4

The Scalable AlphaZero Model

In this section we describe in detail our RL based model for scalable board games. Our model is based on AZ, equipped with additional components that allow it to train on small board sizes and play on larger ones. The board game environment encodes the rules of the game and maintains the board state. We denote by $\mathcal{A}$ the set of possible actions and by $\mathcal{S}$ the set of possible board states.

As mentioned in Section 3.3, the AZ player is an RL model consisting of a combined neural network, $f_{\theta} : \mathcal{S} \rightarrow [0, 1]|\mathcal{A}| \times \{-1, 0, 1\}$, with parameters $\theta$ and an MCTS. The network takes as input the raw board representation of the current state $s \in \mathcal{S}$, and outputs $f_{\theta}(s) = (p_s, v_s)$, where the probability vector $p_s = (p_1, p_2, ..., p_{|\mathcal{A}|}) \in [0, 1]|\mathcal{A}|$ represents the probabilities of selecting each action on the board, and the value $v_s \in [-1, 1]$ estimates the chances of the current player winning the game (i.e., $-1$ for losing, $0$ for a tie and $1$ for winning), given its current state. At each state $s$, an $f_{\theta}$-guided MCTS is activated. The MCTS procedure then outputs the probability $\pi$ for playing each valid move. For a full description of the MCTS procedure, see Appendix 3.3.3. The pseudocode for our model, including the MCTS procedure, is provided in Appendix A.1.

To summarize, the main changes we made to the original AZ are:

- Replacing the CNN by our GNN.
- Adding subgraph sampling for guidance of the MCTS search.
- Removing rotation and reflection augmentations in the training set.

The next sections elaborate on each of these components.

4.1 Replacing the CNN

The main difference between our scalable RL player and AZ comes from choosing the specific neural network type. AZ uses a CNN as the network $f_{\theta}$. As already mentioned, CNN architectures are limited due to the specific input they require, thus they do not enjoy the potential computational benefits of scalable methods. The message passing
technique used in a GNN [GSR+17] (see Section 3.4) allows the network to get a variable sized graph with no limitation on either the number of nodes or the number of edges. In fact, a GNN only requires a fixed size of feature dimension for each node (and each edge, if edge features are used). This last observation makes a GNN a scalable neural network according to the definition above. Consequently, replacing the original CNN in the AZ framework with a GNN is a key step toward our construction of a scalable player mechanism.

To instantiate \( f_\theta \) as a GNN, we first need to translate the board state into a graph. We define the graph \( G(s) = (V,E) \) where nodes in \( V \) are the positions on the board (usually, \( V = \{(i,j)|i,j \in [n]\} \) for a grid-like square board of size \( n \times n \)), and the edges in \( E \) connect “geographically” adjacent positions on the board (for the grid-like example above we connect only vertical and horizontal neighbors and discard diagonal neighbors). For a node \( v \in V \) we denote by \( h_0^v \) the initial feature representing the current piece placed on \( v \) (-1 for a light piece, 1 for a dark piece and 0 for an empty square). Last, we add a dummy node (as demonstrated in [GSR+17]) that is connected to all other nodes in \( V \), allowing us to improve the long-distance data flow between nodes. The dummy node has an initial feature \( h_0^{\text{dummy}} = 0 \). Figure 4.1 illustrates the graph generation procedure, which corresponds to the initial board of the Othello game.

![Figure 4.1](image)

(a) Othello initial board of size 6 × 6  (b) Corresponding graph

Figure 4.1: Illustration of the graph \( G(s_0) \) generated based on the initial board state, \( s_0 \), of the game Othello. The blue and green nodes correspond to the light and dark pieces, respectively. Our additional dummy node is the central red with black border node connected to all other nodes. It is the only node that does not represent any square on the board.

Our GNN receives the generated graph as input and outputs both the probability for playing the specific action corresponding to the node and the value of the current state (i.e., the whole graph). The final GNN architecture, which is based on the GIN model (see the discussion in Section 2.2) with extra skip connections, is illustrated in Figure 4.2.
The architecture was implemented using *PyTorch Geometric* [FL19]. It contains the following modules:

1. Three GIN layers with layer normalization and a ReLU activation function.
2. Concatenation of all previous intermediate representations.
3. Two fully-connected layers with batch normalization, ReLU activation function and dropout.
4. The computation is separated into two different heads, for computing the policy $p$ and the value $v$. $p$ is computed using one fully-connected layer, followed by a log-softmax operation, yielding the probability vector. $v$ is computed using one fully-connected layer, followed by a global mean pooling layer (i.e., the mean among all nodes) and, finally, a tanh nonlinearity function.

![Neural network architecture](image.png)

**Figure 4.2: Neural network architecture**

### 4.2 Guiding MCTS

The second change we made refers to the guidance of the MCTS by the network $f_\theta$. According to MCTS, $f_\theta(s') = (p_{s'}, v_{s'})$ is computed for each nonterminal leaf node $s'$ discovered during the game. These values are used for updating the MCTS variables $P(s', \cdot) = p_{s'}$, propagating $v_{s'}$ along the path seen in the current game simulation, and updating their $Q(s, a)$ values accordingly.

Here we can take advantage of the scalability of our network $f_\theta$, and enhance the performance of the tree search. Upon arriving at $s'$, we sample a few subgraphs of the graph generated by $s'$ and send them to $f_\theta(\cdot)$. For each subgraph we first sample the subgraph size $d \in [(m - 1)^2, m^2]$ and then sample $d$ nodes present in the subgraph. The subgraph size $d$ should be large enough to form an “interesting” new state and include enough legal actions. The subgraphs’ size range $(m)$ as well as the number of sampled subgraphs are two hyperparameters of our model. Note that sending more
than one graph to the network for each newly visited leaf node can be implemented efficiently using batches, which increases the prediction time by only a small factor. Our experiments show that using a small number of subgraphs improves the player’s performance remarkably. Figure 4.3 provides an example of a few sampled subgraphs that correspond to the initial Othello graph from Figure 4.1b.

Figure 4.3: Six random subgraphs of the graph presented in Figure 4.1b. The subgraphs are sent in batches to $f_\theta$, and then the average probability for playing each action is used to guide the MCTS procedure.

The MCTS variables are updated in our model according to

$$P(s', \cdot) = \frac{(p_1 + p_2 + p_1 \odot p_2)}{2}$$

where $p_1$ is the probability vector $p_{s'}$ taken from the evaluation $f_\theta(G(s'))$, $p_2$ is the scatter mean/ max of the probability vectors computed on the subgraphs (i.e., it takes into account how many times a node was sampled), and $\odot$ stands for element-wise multiplication. Propagating $v(s')$ remains unchanged.\(^1\)

4.3 Training Pipeline

The training pipeline, as in the AZ model, comprises a loop between the selfplay and optimization stages. The game result, $z \in \{-1, 0, 1\}$, of each selfplay is propagated to all the states visited during the game. The player plays against itself, thus accumulating positive and negative examples. The neural network parameters are optimized at the end of the selfplay stage to match the MCTS probabilities $\pi$ and the winner $z$. For more details about the AZ training pipeline, see Appendix 3.3.2.

\(^1\)GitHub repository: pytorch_scatter (released under the MIT license)
For each training example produced during selfplay, AGZ generates extra examples by looking at rotations and reflections of the board. In contrast, AZ did not use these extra training examples, demonstrating the strength of their guiding network. By looking at the board as a graph, our GNN takes these invariances into account, thus justifying the removal of extra examples without the need to enhance the performance of the guiding network (e.g., by increasing the number of parameters). Consequently, removing rotation and reflection examples results in a massive reduction in the required training resources and substantially speeds up training time (by 5x).
Chapter 5

Experiments and Results

5.1 Experimental Setup

5.1.1 Board Games

We conduct our experiments on three scalable board games:

1. **Othello** [Lan85]: also known as Reversi. Players alternately place stones on the board trying to “capture” the opponent’s stones. Any straight line sequence of stones belonging to the opponent, lying between the just placed stone and another stone of the current player, are turned over and switch colors. The winner is determined by the majority stones’ color.

2. **Gomoku**: also known as ‘Five in a row’ or Gobang. Players take turns placing stones on the board. The first player to place \( k \) (here 5) stones in a row, a column or a diagonal, wins.

3. **Go**: the well-known game of Go [Smi08]. Two players alternately place stones on intersections of the board with the goal of surrounding more territory than the opponent.

Table 5.1 analyzes the game complexity of the games used for testing. Figure 5.1 includes the boards of the three games as well as a description of their main characteristics.
Table 5.1: Strategic complexity of small/large Othello, Gomoku and Go games given by evaluations of their state and action space size (upper bound).

<table>
<thead>
<tr>
<th></th>
<th>Othello</th>
<th></th>
<th>Gomoku/ Go</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$8 \times 8$</td>
<td>$16 \times 16$</td>
<td>$9 \times 9$</td>
<td>$17 \times 17$</td>
</tr>
<tr>
<td>$</td>
<td>S</td>
<td>$ (states)</td>
<td>262,144</td>
<td>16,777,216</td>
</tr>
<tr>
<td>$</td>
<td>A</td>
<td>$ (actions)</td>
<td>65</td>
<td>257</td>
</tr>
</tbody>
</table>

Figure 5.1: (a) An $8 \times 8$ Othello board. By placing a light stone at the star-marked square, all dashed lines would be colored in white. (b) A Gomoku board. The game’s objective is to place 5 consecutive pieces of the same color either in a row, column or diagonal. (c) A $19 \times 19$ Go board. In the area marked with a red square we can see that the player with the dark pieces occupied the entire territory by surrounding the light pieces from every optional direction.

5.1.2 Training Setup

Hardware

Our RL infrastructure runs over a physical computing cluster. To train SAZ, we use one GPU (TITAN X(Pascal)/PCIe/SSE2) and one CPU (Intel Core i7), referred to as one resource unit. For each experiment conducted, we use the same resources to train. Our Othello player model was trained for three days on boards of all sizes, between 5 and 8. Our Gomoku player was trained for 2.5 days on boards of random sizes, between 5 and 9. Our Go player was trained for three days on boards of random sizes, between 5 and 9.

Hyperparameters

The hyperparameters are selected manually via preliminary results on small boards and then optimized for larger ones. The training parameters for SAZ and the original AZ are presented in Appendix A.2.
5.1.3 Opponents

Reference opponents

We define two reference opponents for each game: a random player that randomly chooses a legal move, and a greedy player that chooses his action based on a hand-coded tactical heuristic score. The specific heuristics for each game is described in Appendix A.2. The greedy opponent provides a sufficient challenge to demonstrate the utility of generalization. Note that both reference players can play on every board size without making any changes to the action-choosing mechanism.

Baseline opponents

For the model analysis we also define some baseline players, each trained for three days (unless otherwise specified), as our model was:

- Model1 refers to training the original AZ (with a CNN replacing the GNN) on the actual board size used for testing. We used a shallower CNN than the one used in the AZ model, due to our limited computational resources (the architecture is described in Appendix A.2). Note that because we failed to train a competitive AZ player with the shallow CNN, we reused symmetries of the training examples (see Section 4.3) as proposed in AGZ model.

- Model2 refers to training SAZ on the actual board size used for testing, rather than smaller boards.

- Model3 is the same player as SAZ without the subgraph sampling component, i.e., the action probabilities are taken directly from the output of \( f_\theta \) on the full graph.

- Model4 is the same as SAZ except here we discard the output of \( f_\theta \) on the full graph; thus, the action probabilities are calculated only according to the sampled subgraphs’ mean.

- Model5 refers to an MCTS guided by a small CNN. The small CNN was trained by the AZ model on a smaller board of size \( m \). The action probabilities are taken as the scatter mean of the network output on all the sub-boards of size \( m \) of the state that is evaluated.

5.1.4 Measure of Success

As a measure of success we use the average outcome of 100 games against one of the reference/baseline opponents, counted as 1 for a win, 0.5 for a tie and 0 for a loss. Each player plays half the time with dark pieces (plays first) and half with light pieces (plays second). We also analyze individually each main change we made. Furthermore, we play against the original AZ player that was trained to play on a large board, which enables
us to measure the effect of our improvements on the training speed and realtime playing performance. Full CNN architecture of the AZ player is described in Appendix A.2. All tables and graphs provided include standard errors (five independent runs).

5.2 Model Analysis

This section provides the full details about each experiment conducted.

5.2.1 The Merits of our Modified Components:

We start with a small ablation study, where we evaluate the contributions of our main changes. We start with the complete SAZ and leave one component out each time, both for training and realtime playing purposes. Note that in this experiment, we focus on the first two changed components presented in Chapter 4. Removal of the third component was tested as well, but, as expected, it has no effect on the performance, as the GNN framework has the property of rotation and reflection invariant. It does, however, increase the training time significantly.

Table 5.2 shows the average outcome (see definition in Subsection 5.1.4) of each model playing against the greedy opponent on a $16 \times 16$ board for Othello, and $17 \times 17$ for Gomoku. Blue and red colors represent whether or not a player wins more than 50% of the games against the greedy opponent. In general, it can be seen that removing each component results in a decrease in performance. Both model1 and model2 produce the poorest results, probably due to insufficient training time on the large board. Model3 is already achieving fair results, while our SAZ slightly improves its performance. We will further discuss the subgraph sampling contribution in the next experiment.

Table 5.2: Leave-one-out study (test average outcome against the greedy opponent)

<table>
<thead>
<tr>
<th>Model</th>
<th>Othello 16 × 16</th>
<th>Gomoku 17 × 17</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAZ [complete model]</td>
<td>0.85 ± 0.02</td>
<td>0.8 ± 0.03</td>
</tr>
<tr>
<td>AZ trained on tested board [model1]</td>
<td>0.44 ± 0.02</td>
<td>0.43 ± 0.04</td>
</tr>
<tr>
<td>SAZ trained on tested board [model2]</td>
<td>0.32 ± 0.03</td>
<td>0.34 ± 0.04</td>
</tr>
<tr>
<td>only full graph [model3]</td>
<td>0.78 ± 0.02</td>
<td>0.73 ± 0.02</td>
</tr>
<tr>
<td>only subgraphs [model4]</td>
<td>0.65 ± 0.01</td>
<td>0.31 ± 0.05</td>
</tr>
</tbody>
</table>
5.2.2 Generalization to Larger Boards

Game Playing Skills

As mentioned, SAZ was designed to allow training and playing on different sizes of input. The generalization study is presented in Figure 5.2 and shows the average outcome (see Subsection 5.1.4) against the reference opponents for Othello and Gomoku, on various board sizes. We also include other baseline players’ performance. All models tested in this experiment were trained for three days on our machine. Overall, SAZ performs significantly better than other methods, consistently winning over 75% of the games against the greedy opponent in all cases.

Among all baseline players, model4 and model5 exhibit the worst performance against both opponents and suffer the greatest performance decrease as the board gets larger. The results of both models suggest that using a small network, applied only on local areas of the full board, does not provide good generalization power, probably because long-term relations are necessary to fully observe the state. Model3 is pretty stable along board sizes, reasonably achieving its best results playing on the board sizes on which it was trained. Observe that our Othello SAZ reaches its peak efficacy on a board size that it had not seen during training.

![Figure 5.2: Average outcome of scalable players against the reference opponents on various board sizes and games. The shadowed areas represent the standard errors (5 independent runs).](image)

We have also examined the generalization ability by considering a slight change in the rules of the game Gomoku. Additional information about this experiment can be found in the supplementary material under Section A.3.

Geometric Perspective

We further examine the generalization power geometrically by considering the GNN actions’ latent space. We constructed synthetic Othello boards of specific form, shown in Figure 5.3a, in different sizes from $6 \times 6$ to $350 \times 350$. We apply Principal Component Analysis (PCA) [WEG87] on the embedding provided by the GNN for two specific actions – one that we consider a “good action” (top-left corner, capturing all opponent...
pieces in the first column) and a second that we deem a “bad action” (bottom-right corner, which does not capture pieces at all).

Figure 5.3b shows the first two components of the PCA analysis of both actions (on the X,Y plane) as a function of the board size (Z axis). Clearly, most of the good actions (blue) are separated easily from the bad ones (red), showing that the latent space successfully encodes the underlying structure of the actions on the board, even for massive board sizes. A 3D plot of the first three PCA components of both actions can be seen in Figure 5.4, with each point representing a board size of $6 \times 6$ up to $350 \times 350$. Except for a few outliers, we can see that the good actions (blue) and the bad actions (red) are located in different areas.

5.2.3 Training Time Analysis

Figure 5.5 shows the progression of our GNN during training. We measure the GNN skill by evaluating the average outcome of model3 (i.e., an MCTS guided by the GNN), at each training stage, against the greedy opponent on a $16 \times 16$ Othello board and a $17 \times 17$ Gomoku board. Since we test the GNN on a larger board than the ones used for training, it can be seen as another measure of the generalization power. As a comparison we train model1 (i.e., original CNN) on the larger boards for 30 days and evaluate it along the training time as well.

We observe that as training advances, model3 gets stronger, achieving around an 80% win rate at the end of training, and reaching parity with the greedy player after a few hours of training. In contrast, to achieve parity, model1 needed between four to five days of training, and achieving model3’s final win rate against the greedy player only after 28 days (Othello) and 23 days (Gomoku).

5.2.4 Comparison to AZ

Table 5.3 shows the average outcome of various scalable players (rows) against the original AZ guided by a CNN (columns). Entries in the table represent the average outcome of the game with respect to the row player. Blue and red colors represent whether or not a specific (row) player wins more than 50% of the games against AZ. The scalable players include our model as well as other baseline players, all trained for three days on small boards (up to $9 \times 9$). AZ players were trained for $\times 10$ days on the large board of the size that was used for testing ($16 \times 16$ or $17 \times 17$).

The results show that SAZ wins all competitions, with a more than 50% win rate on Othello and 100% on Gomoku. Model3, which does not use the subgraph sampling technique, also competes fairly well with AZ, but still reduces the performance by 24% on Othello. Both model4 and model5 Othello players are not competitive compared to AZ, showing again that global dependencies on the board are critical for gameplay.
Figure 5.3: (a) The synthetic Othello boards of increasing sizes we created. A similar board of the same form was created for all board sizes between $6 \times 6$ and $350 \times 350$. (b) The first two principal components of the embeddings provided by our GNN (X,Y plane) as a function of the board size (Z axis). Blue points refer to the embedding of the “good action” of placing a dark piece in the top-left corner. Red ones refer to the “bad action” of placing a dark piece in the bottom-right corner.

Nevertheless, both models produce a positive win rate against AZ on Gomoku, showing that local structures are more helpful for mastering this game. To further illustrate the capabilities of SAZ compared to AZ, we conduct the same experiment with $20 \times 20$ Othello and $19 \times 19$ Gomoku boards. The effect is much stronger, as SAZ wins 84% of Othello games against AZ. The AZ $19 \times 19$ Gomoku player performs poorly in all
Figure 5.4: The first three principal components of the embeddings provided by our GNN. Each point represents one board size between $6 \times 6$ and $350 \times 350$. Blue points refer to the embedding of the “good action” of placing a dark piece in the top-left corner. Red ones refer to the “bad action” of placing a dark piece in the bottom-right corner.

Figure 5.5: Progression of GNN skill along training. The average outcome is evaluated by playing against the greedy opponent on Othello (board size $16 \times 16$) and Gomoku (board size $17 \times 17$).

cases, suggesting that enlarging the board should be accompanied either with a more powerful CNN architecture or with more training.
Table 5.3: Average outcome of scalable players (rows), trained on small boards, against the original AZ players (columns), trained on the tested board size over nearly ×10 more training time.

<table>
<thead>
<tr>
<th>Model</th>
<th>Othello AZ</th>
<th>Gomoku AZ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16 × 16</td>
<td>20 × 20</td>
</tr>
<tr>
<td>SAZ [complete model]</td>
<td>0.54 ± 0.02</td>
<td>0.84 ± 0.01</td>
</tr>
<tr>
<td>only full graph [model3]</td>
<td>0.41 ± 0.01</td>
<td>0.72 ± 0.01</td>
</tr>
<tr>
<td>only subgraphs [model4]</td>
<td>0.05 ± 0.03</td>
<td>0.28 ± 0.04</td>
</tr>
<tr>
<td>small CNN [model5]</td>
<td>0.1 ± 0.03</td>
<td>0.33 ± 0.03</td>
</tr>
</tbody>
</table>

5.2.5 Go Evaluation

Training AZ to the game of Go with full boards is computationally challenging with our available resources. Recall that Deepmind used ~5000 TPUs for 13 days to train AZ 19 × 19 Go player. We therefore trained our SAZ for three days on Go boards of maximal size 9 × 9. To test our model we trained two AZ players on boards of sizes 9 × 9, 15 × 15 and 19 × 19 for 30 days on each size.

Our analysis suggests that SAZ wins around:

- **68%** of the games against AZ on a 9 × 9 board.
- **76.6%** of the games against AZ on a 15 × 15 board.
- **84%** of the games against AZ on a 19 × 19 board.

These results as well as the extensive experiments on Othello and Gomoku, which have some similarity to the properties of Go, indicate that our method can lead to solutions that master the game of Go with much less computational overhead.
Chapter 6

Conclusion and Future Directions

In this thesis we presented an end-to-end RL model for training on and playing scalable board games. Central to our approach is the combination of a scalable neural network (GNN), and the AZ algorithm. The use of GNNs facilitated the enhancement of the model by the subgraph sampling technique, and enabled scaling from small boards to large ones.

Through extensive experimental evaluation, we demonstrated the effectiveness of our method in learning game strategies, which we validated using different games and various board sizes. The generalization analysis suggests that learning on small boards is faster and more practical than learning solely on large boards. The experiments shown in this thesis suggest that SAZ offers a promising new technique for learning to play on large boards, requiring an order of magnitude less training, while keeping the performance level intact.

We have left a number of potential improvements to future work. First, to date we have focused on board games whose actions refer to the nodes on the graph. This focus was natural because GNNs output the feature vector for each node. Nevertheless, we can use the same approach for another family of board games by using GNNs that estimate edge features (e.g., the game of Chess can be formulated as a graph problem where edges correspond to the actions on the board). A promising approach to achieve this could be to use the method of [BKW17] who employ the incident node features to derive edge representations.

Furthermore, our subgraph sampling technique, which effectively improved our model performance in our context by reducing the GNN’s uncertainty, is of potential independent interest. It would be interesting to validate this approach in different domains. Another promising idea would be to use a model pretrained with our approach and then finetune it to a larger board. The finetuned model would possibly enhance the performance on that size. Finally, it would be important to consider deeper GNN architectures, which will possibly enable discovering longer term dependencies on the board.
Appendix A

Appendix

A.1 Pseudocode

Here we provide the pseudocode for our ScalableAlphaZero model. Algorithm A.1 describes the MCTS parameters update starting from an initial state $s$, Algorithm A.2 describes the update rule for the policy vector $\pi$ based on the updated MCTS and Algorithm A.3 describes the training pipeline.

A.2 Global Setup

**GNN:** For Gomoku and Go, we used three layers of GIN with a ReLU nonlinearity and a hidden dimension of 512. With Othello, we used the same architecture but replaced the three layers of GIN with three layers of GCN.

**MCTS:** The number of MCTS simulations was set to $N_{\text{sim}} = 100$. We used $c_{\text{PUCT}} = 1.5$ for the exploration and exploitation parameter. The temperature was set to $\tau = 1$ at the beginning of the tree search and, after 25 search iterations, was changed to $\tau = 0$ (i.e., the action is chosen by argmax). Consider a board of size $n \times n$. The number of sampled subgraphs is larger when the board size increased and is set to round($n/2$). For the parameter that controls the subgraphs size we used $m = n - 1$ or $m = n - 2$. 

41
Algorithm A.1 MCTS\(_\tau\)

Input:

- an initialized MCTS tree \( \mathcal{T} \)
- a state \( s_0 \) (root)
- number of subgraphs to use \( k \)
- size of subgraphs \( m \)

1: \( s \leftarrow s_0 \)
2: if \( s \) is terminal then
3: \( v_s \leftarrow \) get game result of \((s)\)
4: end if
5: \( g \leftarrow \) convert \( s \) to graph
6: while \( s \) is not terminal do
7: if \( s \) is not expanded then
8: \( (p_1,v_s) \leftarrow \text{GNN}(g) \)
9: sample \( k \) subgraphs of size between \((m - 1)^2 \) and \((m + 1)^2 \)
10: \( (p_2,\sim) \leftarrow \text{GNN}(\text{subgraphs}) \)
11: \( Q(s,a) \leftarrow v_s \)
12: \( P(s,\sim) \leftarrow \frac{p_1 + p_2}{2} \)
13: \( N(s,\sim) \leftarrow 0 \)
14: end if
15: \( a \leftarrow \) action that maximizes \( U(s,\cdot) \)
16: \( s \leftarrow \) next state of \( s \) after selecting \( a \)
17: end while
18: while \( s \neq s_0 \) do
19: \( a \leftarrow \) previous action
20: \( s \leftarrow \) previous state
21: \( Q(s,a) \leftarrow \frac{(Q(s,a)\cdot N(s,a)+v_s)}{(N(s,a)+1)} \)
22: \( N(s,a) \leftarrow N(s,a) + 1 \)
23: end while

Output: an updated \( \mathcal{T}_s \).

Algorithm A.2 compute \( \pi_s \)

Input:

- a state \( s \)
- a temperature \( \tau \)
- the number of MCTS simulations \( N_{\text{sim}} \)

1: for \( i = 1 \) to \( N_{\text{sim}} \) do
2: \( \text{MCTS}_\tau(s) \)
3: end for
4: \( \pi(s,\cdot) \leftarrow N(s,\cdot)^{\frac{1}{\tau}} \)

Output: \( \pi_s \)
Algorithm A.3 train

**Input:**

- a maximal board size for training $n$ (squared)
- the number of AZ iterations $N_{\text{iter}}$
- a GNN $f_{\theta}$
- an MCTS $T$
- the number of AZ iterations to include in history $H$

1: for $i = 1$ to $N_{\text{iter}}$ do
2:     sample board size $n_0 \in \{n, n-1, n-2, n-3\}$ w.r.t a constant probability vector
3:     training examples $\leftarrow$ selfplay($n_0 \times n_0$, $f_{\theta}$, $T$)
4:     add training examples to history
5:     if length(history)$>H$ then
6:         pop history
7:     end if
8:     for batch in shuffled history do
9:         $(P_s, V_s) \leftarrow GNN(\text{batch})$
10:        batchloss $\rightarrow$ $\text{MSE}(z(\text{batch}), V_s) + \text{CrossEntropy}(\pi_{\text{batch}}, P_s)$
11:    end for
12: end for
13: compute total loss
14: optimize GNN parameters $\theta$ to minimize total loss

**Output:** optimized GNN $f_{\theta}$
**CNN Architecture:** CNN architecture is relevant to the experiments that include the original AZ player (i.e., model1). It contains the following modules:

1. 2d convolutional layers with 512 channels, a kernel of size three, stride= 1 and padding= 1, followed by 2d batch normalization and a ReLU activation function.

2. 2d convolutional layers with 512 channels, a kernel of size three and stride= 1, followed by 2d batch normalization and a ReLU activation function.

3. A fully-connected layer with hidden dimension of size 1024 and dropout, followed by 1d batch normalization and a ReLU activation function.

4. A fully-connected layer with hidden dimension of size 512 and dropout, followed by 1d batch normalization and a ReLU activation function.

5. The computation is separated into two different heads, for computing the policy $p$ and the value $v$. $p$ is computed using one fully-connected layer from input of size 512 to output of size $|A|$ (number of possible actions), followed by a log-softmax operation, yielding the probability vector. $v$ is computed using one fully-connected layer from input of size 512 to output of size 1, followed by a tanh nonlinearity function.

**Greedy Players Heuristics:** As mentioned in Chapter 5, for our challenging baseline opponent we defined a greedy player, which chooses his actions based on a hand-coded heuristic score. The heuristics are unique for each game: for the game of Othello, the state score is the difference between the player’s stones and those of his opponent; for the game of Gomoku, the score is the length of the maximal sequence of the current players’ stones minus the length of the maximal sequence of opponents’ stones; for the game of Go, the score is evaluated by the difference between the player’s territories and those of his opponent.

**Training and Environment:** Our loss function did not include a regularization term (i.e., $c = 0$). The training set included examples from 20 iterations of selfplay and optimization.

For our multiple-sized SAZ training we randomly sampled a board size at the beginning of each game in the selfplay procedure (see Section 3.3), taken from a probability distribution that is proportional to the board size. For example for training GoMoku we used boards of sizes $(6 \times 6, 7 \times 7, 8 \times 8, 9 \times 9)$ and the probability vector for choosing each size was $(0.4, 0.3, 0.2, 0.1)$. The full algorithm is described in Section A.1.

We used PyTorch Geometric [FL19] for the implementation of the GNN. We used alpha-zero-general for the re-implementation of the AlphaZero model with our modified components,\(^1\) and used the Go environment from alpha-zero-general-with-go-game.\(^2\)

---

\(^1\)GitHub repository: alpha-zero-general (released under the MIT license).

\(^2\)GitHub repository: alpha-zero-general-with-go-game (released under the MIT license).
A.3 Gomoku Generalization Test

In this section, we examine the generalization ability of our model from a different perspective. In our experiments, the winner of the Gomoku game is the player who places five consecutive stones of the same color in a row, a column, or a diagonal. The version we used is also referred to as 'five in a row'. The number of consecutive stones can be changed without affecting any other part of the game. In this experiment, the same SAZ player, which was trained according to the original rules of the game, competed against the greedy player on a $17 \times 17$ board by using different 'k in a row' settings, where $k \in [6,7,\cdots,15]$. We also played against the GNN player (without subgraph sampling) (i.e., model3) and the original AZ player (i.e., model1). Recall that SAZ and model3 were trained for three days on our machine, while AZ was trained for 30 days on the same hardware.

Table A.1: Gomoku playing on various 'k in a row' (test SAZ’s average outcome against the greedy opponent, AZ player and model3)

<table>
<thead>
<tr>
<th>Model</th>
<th>SAZ vs. Greedy</th>
<th>SAZ vs. AZ [model1]</th>
<th>SAZ vs. GNN [model3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k = 6$</td>
<td>0.79</td>
<td>1</td>
<td>0.60</td>
</tr>
<tr>
<td>$k = 7$</td>
<td>0.76</td>
<td>1</td>
<td>0.50</td>
</tr>
<tr>
<td>$k = 8$</td>
<td>0.76</td>
<td>1</td>
<td>0.54</td>
</tr>
<tr>
<td>$k = 9$</td>
<td>0.76</td>
<td>1</td>
<td>0.64</td>
</tr>
<tr>
<td>$k = 10$</td>
<td>0.73</td>
<td>1</td>
<td>0.64</td>
</tr>
<tr>
<td>$k = 11$</td>
<td>0.80</td>
<td>1</td>
<td>0.40</td>
</tr>
<tr>
<td>$k = 12$</td>
<td>0.67</td>
<td>0.97</td>
<td>0.60</td>
</tr>
<tr>
<td>$k = 13$</td>
<td>0.69</td>
<td>0.96</td>
<td>0.52</td>
</tr>
<tr>
<td>$k = 14$</td>
<td>0.57</td>
<td>1</td>
<td>0.515</td>
</tr>
<tr>
<td>$k = 15$</td>
<td>0.63</td>
<td>0.95</td>
<td>0.42</td>
</tr>
</tbody>
</table>

Table A.1 shows the average outcome of SAZ against the greedy player, AZ player and model3, out of 100 games. Blue entries indicated that SAZ won more than half of the games, while red entries indicate that he won less than 50%. As we can see, SAZ wins most of the competitions against each of his opponents. Against the greedy player the results are higher for lower values of $k$. Against model3 the win rate is approximately equal with a slight advantage to SAZ. For the majority of $k$ values SAZ wins around 60% of the games, while only for $k = 11,15$ model3 performs better.

Since the same player is used for different configurations of $k$, we expect him to understand that his goal is to achieve more consecutive stones than his opponent. This is rather than placing a specific number of stones. Therefore, we expect him to adjust his strategies to this mission. The results of this experiment indicate that a GNN can handle slight changes in the rules of the game better and even improve this ability using the subgraph sampling technique.
Bibliography


Showcases of the Decrease in Training Complexity: The scalable AlphaZero model performs better on smaller board games. For instance, the model was trained on a small board game ("Atari"") and then applied to a larger board game ("Shogi") with significantly different rules. The model was able to learn and improve its performance on both games, showing its adaptability and scalability.

Results from the experimental setup show that the model can reach competitive levels of play in a shorter time, despite the increased complexity of the larger board game. This demonstrates the model's ability to learn effectively with limited resources.

Future work in this area will further explore the potential of scalable AlphaZero in various domains, including more complex and diverse games. The results of this research are expected to contribute to the development of more efficient and adaptable AI models for board games and beyond.

In conclusion, scalable AlphaZero represents a significant advancement in the field of AI, offering a promising approach to tackling complex and diverse problems.
תקציר

הﳌידת המשקית לה חכשנה על והתגרו מתוכן ועובר בני אדם ועובר חוקר מדע מתכונת. על משלו, כנף חסותו, העבר גם כן שהחנכה והחנוך, ביכולתם של חסונית והחנה למעט את כל בתיו, לשכת החנכה והחנה מתכונת בהברחת כל החנוכהcherche בנמצא התרשים העם של החנוך, ובעזרת חכמה ומחלקה העם של החנוך, שלחח את עד שמו, באס gamle וברח מחים עולים של חום, בברזים וברחמחים של חמה, בברזים וברחמחים של חמה, בחזקת מחזור של חמה, בברזים וברחמחים של חמה, בחזקת מחזור של חמה.
Shai Ben-Assayag and Ran El-Yaniv. Train on small, play the large: Scaling up board games with alphazero and gnn, Submitted, 2021.
הtteמיא חלק, שחק בקטן:ċאלאלת משחקי
לוח בعزيزת אלפא–זירו ורשתות נוירונים
גרפיט

ה뇨 crater על מחקר

לשם מילוי חלקי של הדרישות ל某一afflesת התואר

מניסור למדעי במדעי המחשב

שי בר-אסיני

הוגש לרגלי התכניות — מכון טכנולוגי לישראל
אב החשפ”א, יולי 2021
התרモノ בניו, שלק בקטן: הכללת משחקי
לוח בעזרת אלפא-지ירו ורשתות נוירוניות
גרופית

שי בונאסיני