Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Espen Bjørge Urheim

Solving *Former* with Machine Learning Techniques

Master's thesis in Applied Physics and Mathematics
Supervisor: Jo Eidsvik
June 2025

NTNU
Norwegian University of
Science and Technology

Espen Bjørge Urheim

Solving *Former* with Machine Learning Techniques

Master's thesis in Applied Physics and Mathematics
Supervisor: Jo Eidsvik
June 2025

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

ABSTRACT

In this thesis, we applied modern machine learning techniques to solve *Former*, a single-player puzzle by NRK (2024). A game of *Former* consists of a 9×7 grid containing 4 repeating shapes. At each turn, the player removes one cluster of identical shapes, and the purpose is to clear the board in as few moves as possible. Through the use of self-made heuristics, supervised learning on self-generated data, and Proximal Policy Optimization, we found effective, though not perfect, strategies for solving the game. To handle the imperfections, we incorporated them in Monte Carlo Tree Search and beam search to efficiently search for solutions to *Former* boards.

Through the combination of machine learning approaches and search techniques, we achieved the best score among any player in Norway on most official boards published by NRK within a short time limit. Across all strategies and search techniques used, we found the best-known solutions to 73% of boards in less than a second and 98% of boards in less than a minute, outperforming any other model previously made, to the best of our knowledge.

SAMMENDRAG

I denne masteroppgaven brukte vi moderne maskinlæringsteknikker for å løse *Former*, et enspillerspill laget av NRK (2024). I *Former* møter spilleren et 9×7 rutenett fylt med fire unike former. For hvert trekk fjerner spilleren én klynge av identiske former, og målet er å fjerne alle fra brettet på færrest mulig trekk. Gjennom bruk av egenutviklede heuristikker, veiledet læring på egengenererte data og "Proximal Policy Optimization" fant vi gode, men ikke perfekte, strategier for å løse spillet. Siden strategiene ikke var perfekte, integrerte vi dem i Monte Carlo-tresøk og strålesøk for å søke etter løsninger på *Former*-brett.

Ved å kombinere maskinlæringsalgoritmer og søketeknikker oppnådde vi de beste resultatene i Norge på offisielle brett publisert av NRK. På tvers av alle strategier og søketeknikker fant vi de beste løsningene på 73% av brettene på under ett sekund og 98% av brettene på under ett minutt, et resultat som, så vidt vi vet, overgår alle andre modeller som har blitt laget for å løse *Former*.

PREFACE

This thesis concludes my Master's degree in Applied Physics and Mathematics. I would first like to thank my supervisor, Jo Eidsvik, for coming up with the brilliant idea of solving *Former* as a machine learning project, for taking the time to assist me and showing great interest in my work, and of course, for his expertise. Thanks to him, I got to do the type of mathematics that interests me the most, on a fun problem that many of my classmates envied me for. Second, I want to thank Matteland, the greatest country in the world, and all my co-students who have spent hours there over the past few years. I hope to return to do some more mathematics in the future.

Espen Bjørge Urheim
Trondheim, June 2025

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Problem Description

In the past few years, daily puzzles have become a global phenomenon. Among the most well-known examples are *Wordle* by the New York Times (2022), *Spotle* by Flatwhite Studios (2024), and *Queens* by LinkedIn (2024). A perhaps lesser-known example is *Former* (Norwegian for "shapes"), released by the Norwegian Broadcasting Company (NRK, 2024). In a *Former* puzzle, players are given a 9×7 board filled with four repeating shapes (Figure 1.1). For each move, players remove a single cluster of identical shapes, and the goal is to clear the board in as few moves as possible. Building larger groups allows them to clear several shapes at once, but this comes at the cost of spending extra moves on building those clusters, which encourages strategic play. The simplicity of the game, combined with the competitive aspect of solving the daily puzzle in fewer moves than everyone else, has caused *Former* to quickly grow in popularity among Norwegians, attracting presumably thousands of players every day.

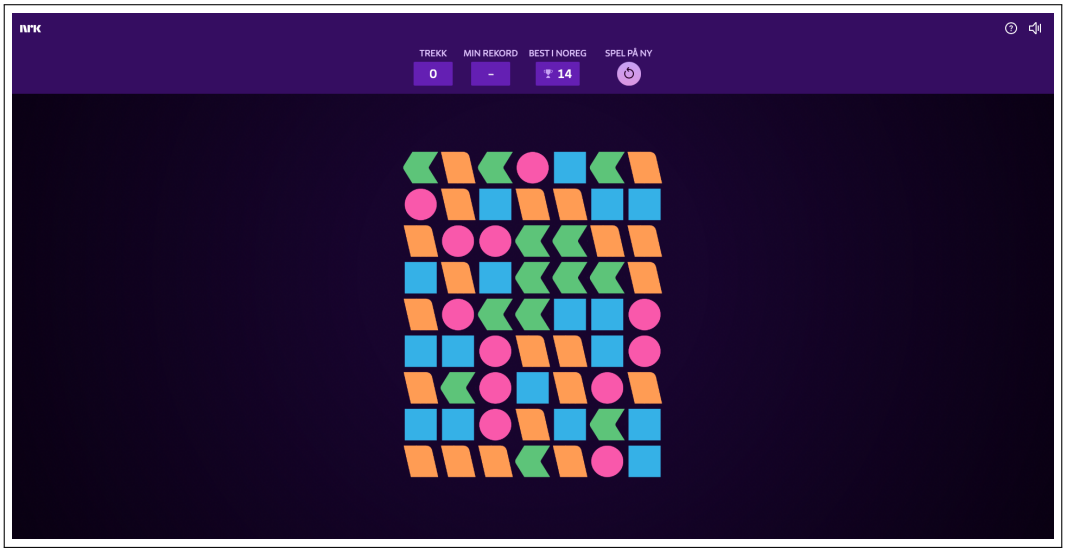


Figure 1.1: The *Former* board published by NRK on June 1st 2025.

Strategic games with a competitive aspect are prime targets for machine learning models. For very small games, where the entire state space can be stored in memory, pure

search techniques alone are sufficient. Russell and Norvig (2016) demonstrate this by solving *Tic-Tac-Toe* optimally with minimax. As the state space of the game grows, however, exhaustive search is infeasible, and the search must be guided by heuristics. Schadd et al. (2008) demonstrate this on the larger single-player puzzle *SameGame*, where Monte Carlo Tree Search (MCTS) combined with game-specific heuristics found good solutions despite the state space being too large for full exploration. For games with astronomically large state spaces, even heuristics struggle to correctly rank moves. For these purposes, self-made heuristics are outperformed by machine-learned strategies: Silver et al. (2018) combined reinforcement learning with MCTS to create AlphaZero, which beat any other player in *Go*, *Chess* and *Shogi*, both humans and machines.

Beyond games themselves, search techniques guided by machine-learned strategies have impact on academic research across areas such as numerical linear algebra and structural biology. Although games are often designed for entertainment, the algorithms used to master them have proven remarkably transferable to scientific problems. For example, the ideas first developed with AlphaZero have since been extended to AlphaTensor, which recently discovered more efficient matrix-multiplication algorithms (Fawzi et al., 2022), and to AlphaFold, which has revolutionized protein structure prediction (Jumper et al., 2021). Thus, although *Former* is primarily an entertainment puzzle, developing machine learning-based methods to solve it contributes to the broader field of modern machine learning research.

In terms of state space complexity, *Former* can be categorized as a large state space game. On a typical initial board, the branching factor is around 36 possible moves, and there are typically more than 10^{21} possible combinations of the first 15 actions. Although this state space is far too large for exhaustive search, Odland (2024) demonstrated that search techniques guided by self-made heuristics can solve many *Former* boards. However, these heuristics alone struggle to generalize across all boards, and they may require tens of minutes of searching on the hardest puzzles. To address this gap, this thesis develops hybrid solvers that combine search techniques with modern machine learning approaches. First, we design a set of simple heuristics that are used in combination with beam search to generate expert data. Then, we use supervised learning techniques to train convolutional policy and value networks on this expert data, to predict promising moves and estimate remaining move counts. In addition to this, we use a state-of-the-art reinforcement learning technique, Proximal Policy Optimization (PPO) (Schulman et al., 2017), to learn strategies through trial and error by playing several million games of *Former*. Once we have trained policy and value networks using these two approaches, we combine them with MCTS and beam search to efficiently look for solutions to *Former* boards. Finally, we compare our solvers with each other, both on randomly generated data and on official NRK test boards. By combining modern machine learning with search techniques, we aim to find the best-known solutions to daily boards published by NRK in less than a minute of searching.

1.2 Main Contributions

With this thesis, we provide the following main contributions:

- **Novel *Former* solvers:** We create 16 solvers in total, 12 of which are based on modern machine learning techniques: 6 that combine MCTS with supervised learning-based policy networks and PPO-based actor networks, and 6 that combine

beam search with supervised learning-based value networks and PPO-based critic networks. These solvers are used to efficiently find solutions to *Former* boards.

- **Code database:** We provide code implementation of *Former* in C++, and Python code for all other components of our solvers. Furthermore, we provide a user interface (see Appendix B.2) that allows anyone to play on any of the 100 official *Former* boards that we have stored throughout the work on this thesis, with solver recommendations. The code is available from the GitHub repository in Appendix B.1, along with a README file that provides further instructions.

1.2.1 Contributions to Sustainability

Solving *Former* with machine learning techniques contributes to the United Nations Sustainable Development Goals (SDGs), albeit indirectly (United Nations, 2015). Here, we highlight a few SDGs, and explain how our work can be considered a contribution to each of them.

- **SDG 4: Quality Education.** We provide a well-documented, open-source code database, which educational institutions, students, researchers or anyone else can use for educational purposes. The work provides explanations and examples of formulating sequential decision-making problems as Markov Decision Processes, designing game heuristics, the use of supervised and reinforcement learning to train policy and value networks, and more. This can be considered a contribution to quality education.

- **SDG 9: Industry, Innovation and Infrastructure.** Although solving *Former* is not a contribution to industry or infrastructure, applying machine learning models to new and entertaining problems stimulates innovation, which is an important part of sub-target 9.5.

- **SDG 17: Partnerships for the Goals.** By publishing our code, data, and pretrained models as open-source information, we encourage collaboration among researchers, educators, and practitioners. This contributes to sub-target 17.6 ("enhance [...] cooperation on and access to science") (United Nations, 2015, "Goal 17" section) by making our methods and results freely available for others to continue the work.

1.3 Structure of Thesis

The thesis is divided into the following chapters:

- Chapter 2 gives a proper introduction to *Former*, where we explain the rules, show gameplay examples, and analyze some interesting properties of the game.

- Chapter 3 provides the background theory of machine learning that is relevant for our approach to solving *Former*.

- Chapter 4 introduces the core principles of search techniques, along with the two algorithms that we use: MCTS and beam search.

- Chapter 5 presents our methodology, where we explain how we combine modern machine learning techniques with search algorithms to solve *Former* boards.
- Chapter 6 displays our results, where we compare our models and solvers to each other and to the best solutions found by any player in Norway.

CHAPTER

TWO

THE *FORMER* GAME

Former is a single-player puzzle made by the Norwegian Broadcasting Organization, NRK (2024). Similar to other popular puzzles such as *Wordle* by the New York Times and *Queens* by LinkedIn, NRK publishes a new puzzle on their webpage every day. Along with it, they reveal the best score that anyone has achieved that day, encouraging players to obtain the best score possible. Our goal in this thesis is to create a solver that can take any board published by NRK as input, and find the best-known solution in less than a minute of searching. Before we dive into how this is done, we need a proper introduction of the game itself. This is the purpose of this chapter.

We first define the rules of *Former* and establish the terminology used throughout the thesis in Section 2.1. Thereafter, in Section 2.2, we explore some gameplay examples, analyze interesting game statistics, and highlight two key properties of the game. Finally, in Section 2.3, we introduce the notation and state the problem of solving *Former* mathematically.

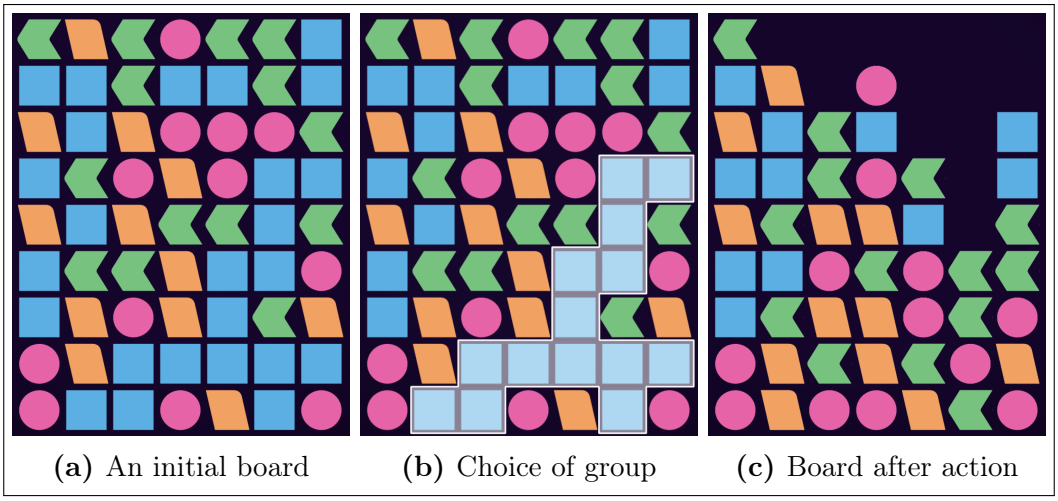
2.1 Rules and Glossary

A *Former* puzzle consists of a 9×7 grid, or a *board*, where each point in the grid contains one of four simple shapes. The goal of the game is to remove all the shapes from the board using a minimum of actions. One action consists of clicking on one of the existing shapes on the board, which removes that shape and all identical ones that are connected to the original shape. After a group has been removed, all columns collapse downward and fill in the empty gap of the removed group. An example of an initial board, an action that causes the removal of a group, and the following collapse of all columns is shown in Figure 2.1.

To describe aspects of the game, we use a variety of terms. For convenience, we therefore include a glossary (Table 2.1) with synonymous terms and the corresponding explanations.

2.2 Game Exploration

To get insight into how *Former* is played, we include two different gameplay examples on the board published by NRK on March 17$^{\text{th}}$ 2025. After these, we study the distribution of shapes in official *Former* boards and approximate the exponential growth in the number



(a) An initial board **(b)** Choice of group **(c)** Board after action

Figure 2.1: **(a)** An initial board in the *Former* game. **(b)** If we click on any shape in the highlighted group, that group is removed and the columns collapse, leading to **(c)** a new board (NRK, 2024).

Shape / color / grid point	In the official board by NRK, there are 4 shapes, each with a corresponding color.
Group	A set of identical shapes that are connected to each other and form a cluster.
State / board / grid	The configuration of shapes. The standard board size is 9×7.
Action / move / turn	The process of clicking on a group of shapes, as illustrated in Figure 2.1.
Agent / player	The individual playing the game. We often use *agent* when referring to a machine.
Model	An agent used to make predictions based on a given board. Models are either self-made heuristics or neural networks.
Solver	A search technique combined with a self-made heuristic or neural network, that solves a board through strategic search based on the incorporated model.

Table 2.1: The terminology used to describe aspects of *Former*.

of action combinations. Finally, we explain two important properties of the game which are crucial for the choice of methodology.

2.2.1 Gameplay Examples

The first gameplay example shows optimal play, according to the best-known solution in Norway. That is, it clears the board in the fewest known number of moves possible. The second is an example of suboptimal play following a greedy strategy.

The best-known solution of the board published by NRK on March 17$^{\text{th}}$ used 11 moves. An example of such a solution is displayed in Figure 2.2. An interesting insight from this gameplay is that the best-known solution starts with removing smaller groups in order to create larger ones that can be removed at later times. For example, in the four first displays in the top row of Figure 2.2, smaller groups were removed to create one large group of orange shapes. This large group is then removed in the fifth action.

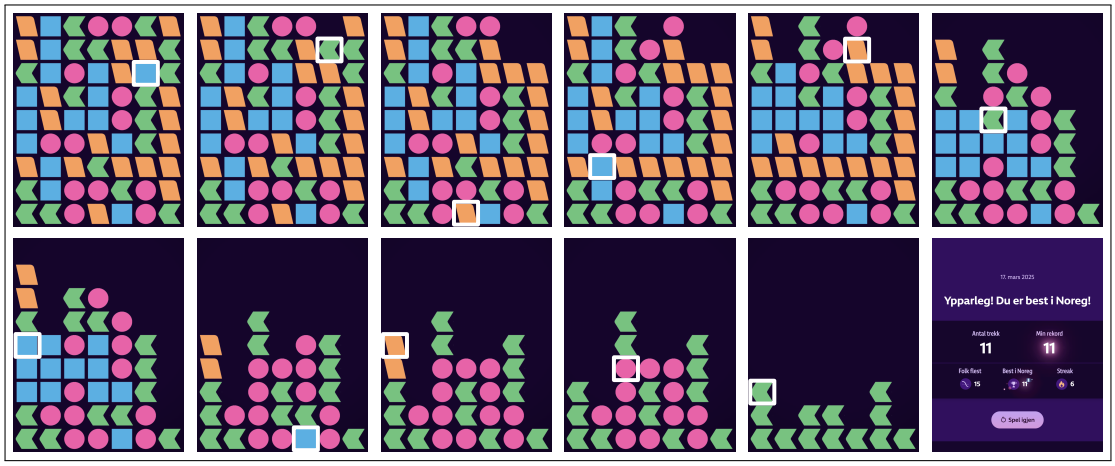


Figure 2.2: Example of optimal play according to the best-known solution of the board from NRK on March 17$^{\text{th}}$ 2025. After the board is cleared, we are congratulated on achieving the best result in Norway. Statistics such as the most common score of all players (15 moves) and the current personal playing streak (6 days) are also shown.

In Figure 2.3, we show an example of suboptimal play, that is, a play-through that uses more moves than the best-known solution. For illustration purposes, we use the greedy strategy of always removing the largest group on the board. This causes many shapes to be removed during the first few steps, but leaves plenty of smaller groups towards the end, resulting in a total of 22 actions used to clear the board. This example illustrates how a poor choice of strategy may lead to a surprisingly large increase in the number of moves used to clear the board compared to optimal play.

2.2.2 Descriptive Statistics

We explore *Former* further by analyzing some interesting statistics of the game. First, we investigate how NRK generates the official *Former* boards, before we generate boards ourselves and study the branching factor of possible actions.

In Figure 2.4 we show the distribution of shape counts for each of the four shapes, based on 100 daily boards published by NRK. We observe that the distributions are quite similar, with nearly equal means and comparable standard deviations. There are some

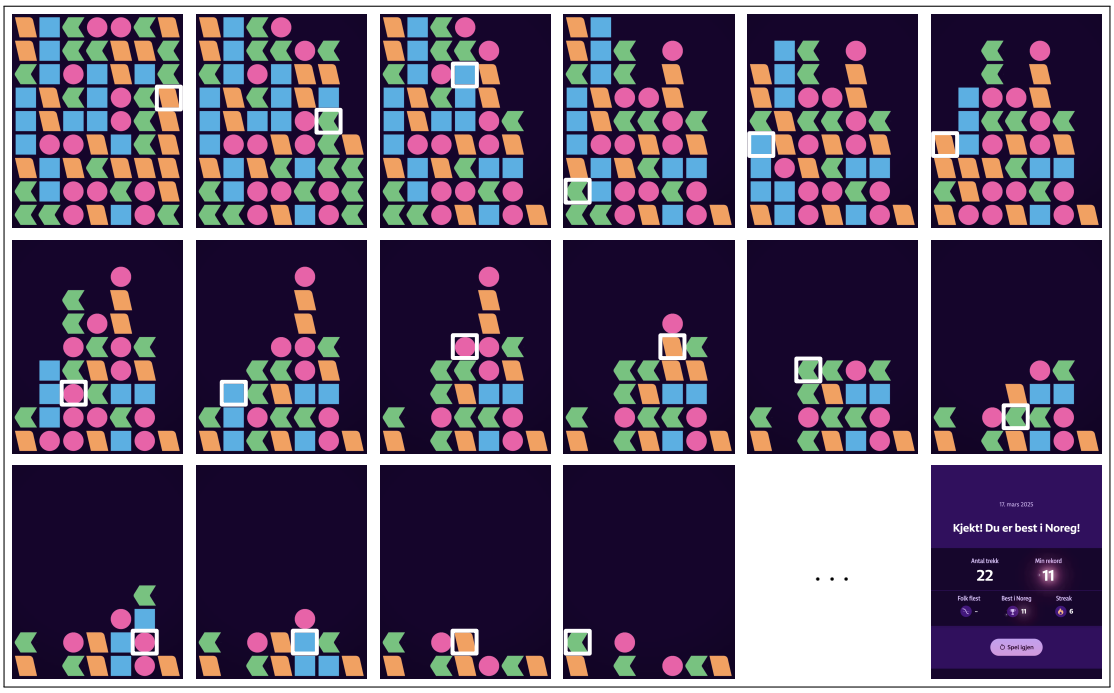


Figure 2.3: Example of suboptimal play following a greedy strategy on the March 17th 2025 board from NRK. The strategy used 22 moves.

outliers, especially for the orange shapes, but this is to be expected from a relatively small set of 100 boards. Overall, the comparable distributions suggest that NRK sample from a discrete uniform distribution when generating their daily boards. For the remainder of this thesis we assume this to be the case, and use a uniform distribution to generate boards ourselves. Due to the importance of this assumption, however, we also perform hypothesis tests to validate it. To keep this section concise, we put the hypothesis tests in Appendix A.

Next, we analyze the size of the action space of *Former*. We begin by generating 1 000 random initial boards, where each of the 9×7 cells is assigned one of the four shapes by sampling from a discrete uniform distribution. For each board, we then simulate 20 random playouts of length 15 moves. During each playout, at move depth $t \in \{0, 1, \ldots, 15\}$ we record the branching factor $G(s_t)$, that is, the number of groups on the board. Pooling over all boards and playouts gives us a large sample of branching-factor values at each depth. From these samples we compute the 10th, 50th (median), and 90th percentiles at each t. Finally, we take the cumulative product of the medians,

$$B_t = \prod_{i=0}^{t} \text{median}(G(s_i)),$$

to estimate the total number of possible action sequences up to depth t. We plot B_t along with the 10th and 90th percentiles, shown in Figure 2.5. Here, we observe the exponential growth in the number of possible action combinations as a game progresses. In an initial board, the median number of possible moves is $B_0 = 36$, and the total number of combinations typically exceeds 10^{21} after 15 moves.

It is important to note that, in these calculations, we have not taken into account sequences of actions leading to the same state, which we call *state collisions*. For example, from the first board of Figure 2.3, it does not matter whether we click top left and then top

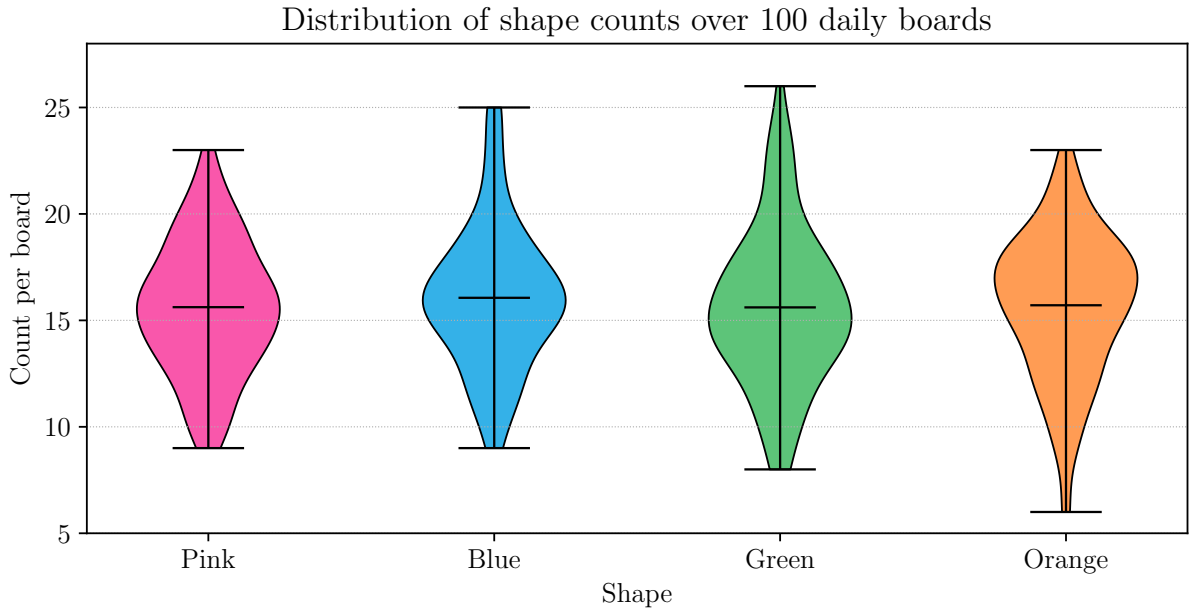


Figure 2.4: The observed distributions of number of shapes, based on 100 daily boards saved from the website of NRK. The middle horizontal lines are the average number of each shape across all boards, whilst the upper and lower lines are the maximum and minimum observed counts, respectively.

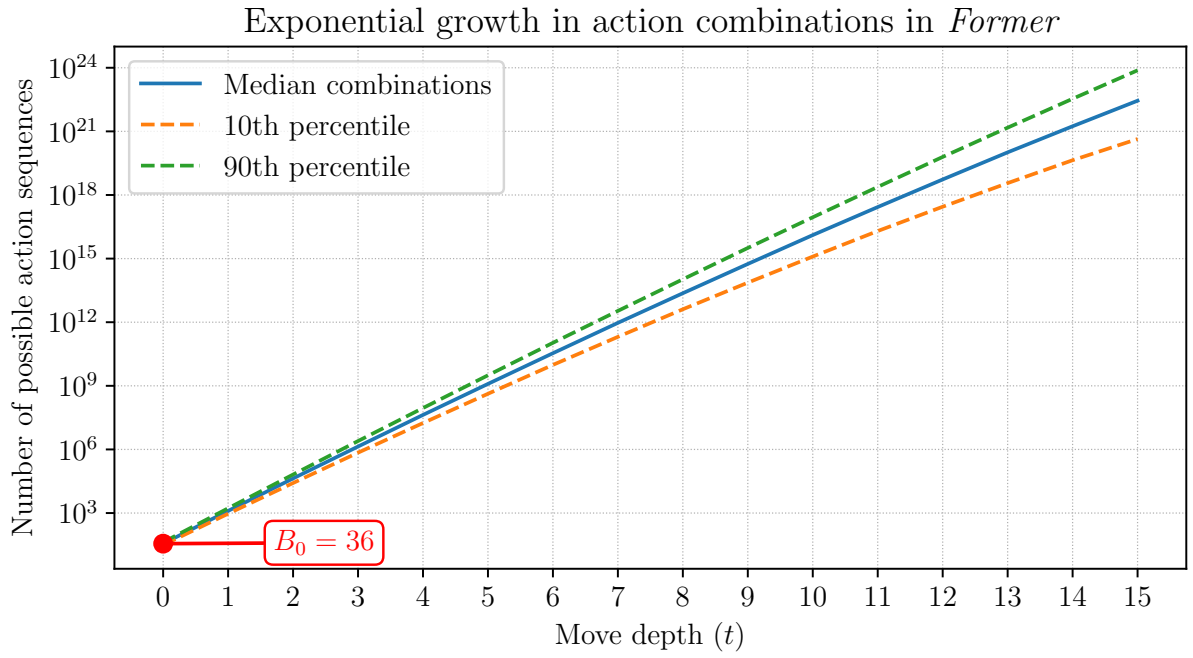


Figure 2.5: The median number of possible action sequences after t moves, along with the 10^{th} and 90^{th} percentiles. The values are estimated empirically based on 20 000 random playouts over 1 000 randomly generated boards. The median number of possible moves from an initial board is $B_0 = 36$.

right, or top right and then top left: both choices lead to the same state. As a result, the branching factor is in reality smaller than suggested by Figure 2.5. Based on simulations over 10 000 boards, we found that on average 41% of the possible combinations of two actions from an initial board lead to a state collision, suggesting that the state space in theory can be reduced by almost half. However, in practice, there is no easy way to perform this state space reduction due to the particular dynamics of the *Former* game: removing a group may completely alter other groups, hence we cannot tell what combinations of actions lead to a state collision without first performing the actions and then comparing the new state to all previously explored states. It is simpler and more computationally efficient to explore the full search space to begin with and not to bother reducing it to its minimal theoretical size. Thus, although we acknowledge that the branching factor in reality is smaller than illustrated in Figure 2.5, this is the branching factor that we are working with in practice.

2.2.3 Important Properties

There are in particular two properties of the *Former* game that are important. Firstly, it is deterministic and, secondly, as discussed, it has a very large state space. In this section, we explain both properties and study how they affect the choice of methodology.

Former is a deterministic game, which means that all the information needed to find the best action given some state is available to the player at all times. There is no randomness involved in an action, and there is no opponent to play against. Consequently, when performing an action, the player can deduce what the next board will look like, what the one thereafter will look like, and so on. As a result, it is theoretically possible to find the shortest sequence of actions to clear the entire board once the initial board is revealed.

Despite being deterministic, it is in practice impossible to solve *Former* by testing all possible combinations of actions due to the large state space of the game. As we observed in Figure 2.5, after $d = 15$ moves of random search, there are approximately 10^{21} possible combinations of actions to discover. Consequently, it is not feasible to explore the entire state space in order to find the optimal solution.

Many other well-known games are deterministic but still extremely difficult to solve due to large state spaces. Multiplayer examples include *Go* and *Chess*, with approximate average branching factors of 35 (Shannon, 1950) and 250 (Silver et al., 2016), respectively. Single-player games include the *15-puzzle* and *Rubik's Cube*, the latter having an approximate branching factor of 13 (Korf, 1997). To handle games with large state spaces, combinations of clever search algorithms and machine learning techniques have proven useful. For example, DeepCubeA combines A^* search with deep neural networks to solve any initial configuration of *Rubik's Cube* (Agostinelli et al., 2019). Due to the somewhat comparable traits of *Former*, we hypothesize that we can use similar methods for our purpose.

2.3 Mathematical Problem Formulation

The purpose of *Former* is to find the shortest sequence of actions to clear some initial board. This makes it a sequential decision-making problem. A common framework for such problems is a Markov decision process (MDP), where we model each play-through as a sequence of board states and actions, with each action leading to a new state according to some transition function. The main assumption underlying the MDP framework is that

the probability of transitioning from one state to another depends only on the current state and the current action performed by the player (Puterman, 1990), which is the case for *Former*. In Section 2.3.1 we define the components of the MDP, and in Section 2.3.2 we state the formal optimization problem for solving the game.

2.3.1 Markov Decision Processes

We model each play-through of *Former* as an episodic, deterministic MDP with a finite horizon. An episode begins at the initial board $s_0 \in \mathcal{S}$ and terminates at the first time step T where all shapes are removed. Formally, the MDP we use to model the game is the tuple

$$(\mathcal{S}, \{\mathcal{A}_s\}_{s\in\mathcal{S}}, P, R, \gamma, T)\,,$$

where:

- The state space $\mathcal{S}$ is the set of all possible 9×7 board configurations.
- The action space for a given state s is $\mathcal{A}_s = \{1, \ldots, G(s)\}$, where $G(s)$ is the number of groups, and thus also the number of unique actions, in state s. If the board is empty, we have reached the terminal state s_T, and the action space is the empty set, $\mathcal{A}_{s_T} = \varnothing$.
- The transition probability $P(s' \mid s, a)$ is deterministic, and given by

  $$P(s' \mid s, a) = \begin{cases} 1 & \text{if } s' = \tau(s, a), \\ 0 & \text{otherwise.} \end{cases}$$

  Here, $\tau(s, a)$ is the transition function, which removes the group belonging to point a in s and collapses the columns, leaving us in s'.
- The reward function $R(s, a) = -1$ is meant to penalize each move equally.
- We set the discounting factor γ to one, so that the total return from one play-through is defined as

  $$G_0 = \sum_{t=0}^{T-1} R(s_t, a_t) = -T, \tag{2.1}$$

  and maximizing G_0 is equivalent to minimizing the number of moves.
- The horizon T is the (random) number of moves until the board is cleared.

2.3.2 Formal Problem Statement

Let π be a decision-making strategy, or a *policy*,

$$\pi : \mathcal{S} \times \mathcal{A}_s \;\to\; [0, 1], \quad \pi(a \mid s) \;\geq 0, \quad \sum_{a\in\mathcal{A}_s} \pi(a \mid s) = 1 \quad \forall s \in \mathcal{S}.$$

That is, π gives the probability of choosing a specific action given some state. For ease of notation, we let

$$\pi(s) = \{\pi(a \mid s)\}_{a\in\mathcal{A}_s}$$

denote the probability distribution over all possible actions from state s. Under π, at each step t the agent samples

$$a_t \sim \pi(s_t), \quad s_{t+1} = \tau(s_t, a_t).$$

Then, solving *Former* means that we find the policy that maximizes the expected return from any state s,

$$\pi^*(s) = \arg\max_\pi E_\pi\big[G_0 \mid s\big] = \arg\min_\pi E_\pi\big[T \mid s\big],$$

where $E_\pi[\cdot]$ denotes the expectation given that the agent acts according to π. Or, equivalently, solving *Former* can be stated as finding the optimal value function $v^* : \mathcal{S} \to \mathbb{R}$, which gives the number of moves remaining if the agent acts according to the optimal policy,

$$v^*(s) = \min_\pi E_\pi[T \mid s].$$

In practice, it is impossible to find π^* and v^*. Therefore, we approximate them and handle the approximation flaws by incorporating them in search techniques. Thus, our pipeline for solving *Former* consists of two steps:

1. Approximate π^* and v^*.
2. Combine the approximations with search techniques to cleverly look for solutions to a given *Former* board.

In the next two chapters, we introduce the theory used for the two respective steps. First, in Chapter 3, we develop the fundamentals of machine learning that we use to approximate π^* and v^*. We also use self-made strategies for this purpose, but these do not require much theory, so we introduce these in the context of the specific methodology. Then, in Chapter 4, we turn our attention to search techniques and how these combine with approximate policy and value functions. Once the proper fundamentals are established, we return with an in-depth explanation of the two-step methodology in Chapter 5.

CHAPTER

THREE

MACHINE LEARNING BACKGROUND

Recall that solving *Former* can be defined as finding an optimal policy function π^* or an optimal value function v^*. One way to approximate these is to identify strategies ourselves, which we call *self-made heuristics*, and then define policy and value functions based on them. Although such heuristics often perform well in solving single-player games, they may not be generalized across all boards and may lead to significant runtime (Browne et al., 2012). A more systematic alternative is to *learn* strategies from data using machine learning techniques, specifically by training deep neural networks.

In this chapter, we develop the theoretical foundations for two approaches to learn a strategy from data: supervised and reinforcement learning. Supervised learning is based on labeled data that directly indicate the correct action or number of moves remaining for a given state, whereas reinforcement learning finds strategies by itself through trial and error. The goal with both approaches is to obtain neural networks that approximate π^* and v^*, which we later use in search techniques to efficiently look for solutions to *Former* boards.

We start by introducing neural networks in Section 3.1, with a focus on the network architectures that we use and how networks are trained. Then, in Sections 3.2, we develop the core principles of supervised learning, and finally, in Section 3.3, we provide the theory for the reinforcement learning algorithm that we use: PPO.

3.1 Neural Networks

Mathematically, a neural network is a function,

$$f = f(\mathbf{x}; \boldsymbol{\theta}),$$

which depends on a set of parameters $\boldsymbol{\theta}$. The purpose of the network is to approximate some other function,

$$f^* = f^*(\mathbf{x}),$$

which we do by tuning $\boldsymbol{\theta}$ so that f approximates f^* as accurately as possible. Here, $\mathbf{x} \in \mathbb{X}$ denotes the tensor representation of a state s, which is what we in practice feed into the neural network. For consistency, we introduce the mapping $\psi : \mathcal{S} \to \mathbb{X}$, which gives a one-to-one relationship between the state s and the tensor representation,

$$\mathbf{x} = \psi(s) \quad \Leftrightarrow \quad \psi^{-1}(\mathbf{x}) = s.$$

Then, for ease of interpretation, we write $f(s;\boldsymbol{\theta})$ to denote the evaluation of state s using the neural network, with the understanding that $f(s;\boldsymbol{\theta}) \equiv f(\psi(s);\boldsymbol{\theta})$.

The function we wish to approximate is either the optimal policy function π^* or the optimal value function v^*. We therefore divide the networks we train into two separate classes: policy networks,

$$f_\pi : \mathcal{S} \to \mathbb{R}^{G(s)},$$

which return a probability distribution over each of the $G(s)$ possible actions from state s, and value networks,

$$f_v : \mathcal{S} \to \mathbb{R},$$

which predict the minimum number of moves remaining to clear some board.

Regardless of whether we train f_π or f_v, the neural network must be able to recognize spatial patterns in a *Former* board, which has a grid-like structure. This makes convolutional neural networks (CNNs) the most reasonable choice (O'Shea & Nash, 2015). In this section, we first define the key building blocks of a CNN, and explain how they are combined to form policy and value networks. Thereafter, we give a brief overview of the neural network training procedure that is common for both supervised and reinforcement learning.

3.1.1 Building Blocks

The overall structure of a CNN is similar to that of any other neural network: It consists of several *layers*, each represented by a function

$$f^{(l)} = f^{(l)}\left(\mathbf{x}_1, \ldots, \mathbf{x}_{w_{l-1}}; \boldsymbol{\theta}^{(l)}\right), \quad l = 1, \ldots, \ell,$$

where ℓ is the number of layers and w_{l-1} is the number of inputs into layer l. The number of inputs w_{l-1} and outputs w_l may vary from one layer to another, hence the index l. When we evaluate some state using the network, we feed the input through the first layer, use the output as input in the next layer, and so on,

$$f(s;\boldsymbol{\theta}) = \left(f^{(\ell)} \circ \cdots \circ f^{(1)}\right)(s;\boldsymbol{\theta}). \tag{3.1}$$

The CNNs we use in this thesis are made up of three main components, each with a different purpose: convolutional blocks, global average pooling (GAP) layers, and fully connected layers. Figure 3.1 shows an example of a value network that combines each of these. In this section, we cover the three main component separately, before we give a brief note on the use of activation functions in the output layer.

Convolutional Blocks

A convolutional block consists of three components in successive order: a convolutional layer, followed by batch normalization and then the ReLU activation function, as illustrated by the colored stacks in Figure 3.1.

The general idea behind a convolutional layer is to extract spatial features from grid-like input by using the convolution operation. Hence, the outputs of a convolutional layer are typically called *feature maps*, to emphasize that they are images representing spatial features from the original input.

A convolutional layer consists of w_l filters, each of which has w_{l-1} kernels. These are $k \times k$ matrices used to perform the convolutions with each of the w_{l-1} feature maps from

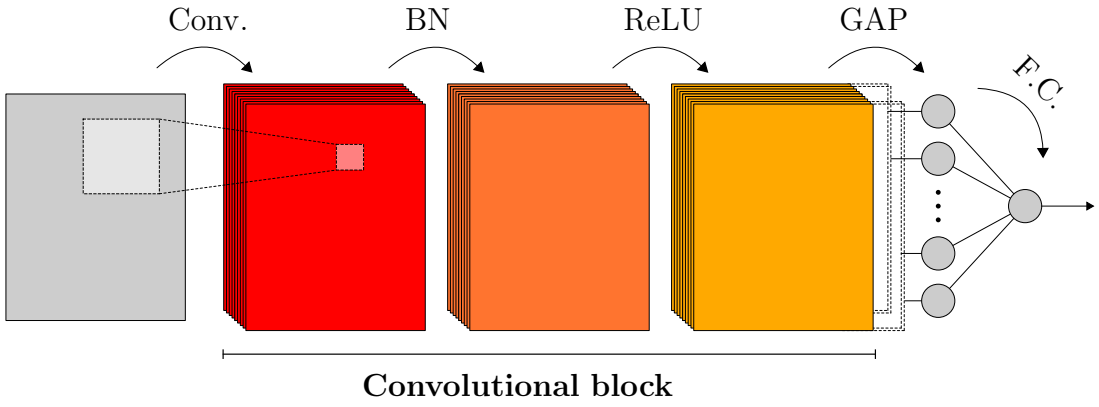


Figure 3.1: An example CNN architecture of a value network. The initial board is first fed through a single convolutional block, where the grid is convolved, fed through batch normalization and ReLU, which after each step outputs a set of feature maps. Thereafter, we employ GAP to reduce each feature map to a scalar value, which in turn are fed into a fully connected layer that outputs a value estimate.

the previous layer. Before convolving, the feature maps are padded with $\lfloor k/2 \rfloor$ zeros on all sides. This ensures that the features at the board edges are convolved just like those in the interior, allowing the output dimensions to be the same as the input. After convolving each feature map with its respective kernel, we sum over all instances and add a bias, which gives a measure of the presence of that particular feature in the input feature maps. To be precise, we define a function $f_j^{(l)} : \mathbb{X}^{w_{l-1}} \to \mathbb{X}$ representing filter j of layer l as

$$f_j^{(l)}(\mathbf{x}_1, \ldots, \mathbf{x}_{w_{l-1}}; \boldsymbol{\theta}_j^{(l)}) = \sum_{c=1}^{w_{l-1}} K_{j,c}^{(l)} * \mathbf{x}_c \; + \; b_j^{(l)}. \tag{3.2}$$

Here, $\boldsymbol{\theta}_j^{(l)}$ contains the parameters that define the kernels $K_{j,c}$ and the bias $b_j^{(l)}$. The bias is a scalar value that is added element-wise. The kernel size k is the same for all filters in the same layer, but it may vary between layers. For games, for example, it is common to use a larger kernel size for the first convolutional layer, say $k = 5$ or $k = 7$, to capture broader spatial patterns, and $k = 3$ for subsequent ones (Silver et al., 2016).

After applying $f_j^{(l)}$ to all feature maps from the previous layer, we perform batch normalization. The general idea behind this is to normalize the feature maps to have zero mean and unit variance, which can accelerate and stabilize the training of deep neural networks (Ioffe & Szegedy, 2015).

After batch normalization, we apply the ReLU activation function, which is used to introduce non-linearity to the network function, since the objective function f^* typically is not linear. The ReLU is defined as

$$\sigma(\mathbf{x}_i) = \max(0, \mathbf{x}_i), \tag{3.3}$$

where the maximum operation is performed element-wise. Thus, the function $f^{(l)} : \mathbb{X}^{w_{l-1}} \to \mathbb{X}^{w_l}$ that represents a full convolutional block is defined by

$$f^{(l)}(\mathbf{x}_1, \ldots, \mathbf{x}_{w_{l-1}}; \boldsymbol{\theta}^{(l)}) = \left\{ \left(\sigma \circ \mathrm{BN} \circ f_j^{(l)} \right) (\mathbf{x}_1, \ldots, \mathbf{x}_{w_{l-1}}; \boldsymbol{\theta}_j^{(l)}) \right\}_{j=1}^{w_l},$$

with σ defined in (3.3), BN denoting batch normalization, $f_j^{(l)}$ defined in (3.2), and $\boldsymbol{\theta}^{(l)}$ containing the parameters that define layer l.

Global Average Pooling

A global average pooling (GAP) layer is used to transition from w_l feature maps to w_l scalar values, $f^{(l)} : \mathbb{X}^{w_l} \to \mathbb{R}^{w_l}$. This is done by taking the global average of each feature map,

$$f^{(l)}(\mathbf{x}_1, \ldots, \mathbf{x}_{w_l}) = (\text{mean}(\mathbf{x}_1), \ldots, \text{mean}(\mathbf{x}_{w_l}))\,,$$

which does not require any parameters. Although fully connected layers can be used for the same purpose, the lack of parameters means that GAP avoids the problem of overfitting, which is often an issue when using a fully connected layer to transition from feature maps to scalars (Lin et al., 2013).

Fully Connected Layers

Fully connected layers are perhaps the simplest layer type, which is typically used in the final layers of a CNN. It consists of *neurons*, and at each neuron, we calculate a weighted average of scalar values $\mathbf{x} \in \mathbb{R}^{w_{l-1}}$ from the previous layer,

$$f_j^{(l)}\left(\mathbf{x}; \boldsymbol{\theta}_j^{(l)}\right) = \sum_{c=1}^{w_{l-1}} W_{j,c}^{(l)}\, x_c + b_j^{(l)}.$$

Here, $W_{j,c}^{(l)}$, $c = 1, \ldots, w_{l-1}$ are the weights associated with neuron j in layer l, and $b_j^{(l)}$ is the bias. Thus, a fully connected layer is represented by the function $f^{(l)} : \mathbb{R}^{w_{l-1}} \to \mathbb{R}^{w_l}$ defined as

$$f^{(l)}\left(\mathbf{x}; \boldsymbol{\theta}^{(l)}\right) = \left\{ \sum_{c=1}^{w_{l-1}} W_{j,c}^{(l)}\, x_c + b_j^{(l)} \right\}_{j=1}^{w_l}.$$

In practice, this is calculated using matrix multiplications, with the notation

$$f^{(l)}\left(\mathbf{x}; \boldsymbol{\theta}^{(l)}\right) = W^{(l)}\,\mathbf{x} + \mathbf{b}^{(l)}, \qquad \boldsymbol{\theta}^{(l)} = \{W^{(l)}, \mathbf{b}^{(l)}\}.$$

Output Activation Functions

Activation functions typically have two purposes in a neural network: (1) to introduce non-linearity, as is the case with the ReLU function (3.3), and (2) to map the output of the network to the correct range. For the second purpose, the choice of activation function depends on whether we have a policy or a value network. In a policy network, the output $\boldsymbol{y} = (y_1, \ldots, y_C)$ needs to be a probability distribution over C possible actions, which is obtained using the softmax activation function,

$$\sigma(\boldsymbol{y}) = \left\{ \frac{e^{y_c}}{\sum_{j=1}^{C} e^{y_j}} \right\}_{c=1}^{C}. \tag{3.4}$$

A value network predicts a scalar value y, for which it is common to use the linear activation function,

$$\sigma(y) = y. \tag{3.5}$$

3.1.2 Training a Neural Network

Now that we have provided the building blocks of CNNs, we switch focus to how they are trained. Recall that the purpose of training a neural network is to tune the parameters $\boldsymbol{\theta}$ so that the network f approximates some target function f^* as accurately as possible. This is an iterative procedure that requires three components:

1. A dataset split into batches

$$B_{\text{train}} = \{(s_i, f^*(s_i)\}_{i=1}^{n_{\text{batch}}},$$

   where $f^*(s_i)$ is the target output for some state s_i.

2. An objective function, $J(\boldsymbol{\theta}; B_{\text{train}})$, that indicates how much the network output deviates from the target output under the current set of parameters $\boldsymbol{\theta}$, based on network evaluations on all samples in B_{train}.

3. An optimization algorithm used to calculate how $\boldsymbol{\theta}$ should be updated to better approximate the target function, by minimizing $J(\boldsymbol{\theta}; B_{\text{train}})$.

The first two points differ between supervised and reinforcement learning. Therefore, we return to the details of these in Sections 3.2 and 3.3. Here, we focus on the shared component, point 3: how $\boldsymbol{\theta}$ is updated.

Updating the parameters of a neural network is an optimization problem. We wish to minimize the objective function $J(\boldsymbol{\theta}; B_{\text{train}})$ with respect to the parameters $\boldsymbol{\theta}$, which requires:

1. computing the gradients of the objective function with respect to the parameters of each layer, $\nabla_{\boldsymbol{\theta}^{(l)}} J(\boldsymbol{\theta}; B_{\text{train}})$, $l = 1, \ldots, \ell$, and

2. an optimization algorithm that uses those gradients to update $\boldsymbol{\theta}$, iteratively.

We obtain the gradients through *backpropagation*, a core algorithm in neural network training procedures (Goodfellow et al., 2016). Backpropagation works by first calculating the error at the output layer and then using the chain rule to propagate the error backwards to the earlier layers. This way, we find an estimate for how much each layer contributes to the overall error of the network output, which we use to update the parameters of each layer. For illustrative purposes, we show the mathematics behind the backpropagation algorithm for fully connected layers. The same principles apply to all other layer types as well (Rumelhart et al., 1986).

Let

$$\mathbf{x}^{(0)} = \big(\psi(s_1), \ldots, \psi(s_{n_{\text{batch}}})\big)$$

be the input states in batch B_{train}, and for each layer l define

$$\mathbf{z}^{(l)} = W^{(l)}\,\mathbf{x}^{(l-1)} + \mathbf{b}^{(l)}, \quad \mathbf{x}^{(l)} = \sigma\big(\mathbf{z}^{(l)}\big).$$

That is, $\mathbf{z}^{(l)}$ is the output of layer l before applying any activation function. After performing the forward pass defined in (3.1) on all samples in B_{train} to obtain $\mathbf{x}^{(\ell)}$ and computing the objective function $J = J(\boldsymbol{\theta}; B_{\text{train}})$, we calculate the gradient with respect to the pre-activation output using the chain rule,

$$\boldsymbol{\delta}^{(\ell)} = \nabla_{\mathbf{z}^{(\ell)}} J = \nabla_{\mathbf{x}^{(\ell)}} J \;\odot\; \sigma'\big(\mathbf{z}^{(\ell)}\big),$$

where $\odot$ is the element-wise multiplication operation. This gradient is interpreted as the change in the objective function given a small change in the pre-activation output. From here, we propagate that error backwards to previous layers using the chain rule,

$$\boldsymbol{\delta}^{(l)} = \big(W^{(l+1)}\big)^\top \boldsymbol{\delta}^{(l+1)} \odot \sigma'\big(\mathbf{z}^{(l)}\big), \quad l = \ell - 1, \ldots, 1.$$

Using this, it can be shown (Goodfellow et al., 2016) that the gradients with respect to the weights and biases of each layer are

$$\nabla_{W^{(l)}} J = \boldsymbol{\delta}^{(l)} \left(\mathbf{x}^{(l-1)}\right)^\top,$$
$$\nabla_{\mathbf{b}^{(l)}} J = \boldsymbol{\delta}^{(l)} \mathbf{1}_{n_{\text{batch}}}.$$

Here, we recall that the full set of parameters $\boldsymbol{\theta}$ contains parameters for all layers, which with the notation of fully connected layers is written as

$$\boldsymbol{\theta} = \left\{W^{(l)}, \mathbf{b}^{(l)}\right\}_{l=1}^{\ell}.$$

Then, $\nabla_{\boldsymbol{\theta}} J$ is the vector formed by stacking the gradients of the weights and biases from each layer,

$$\nabla_{\boldsymbol{\theta}} J = \left\{\nabla_{W^{(l)}} J, \nabla_{\mathbf{b}^{(l)}} J\right\}_{l=1}^{\ell}.$$

To update $\boldsymbol{\theta}$ based on these gradients, we use the adaptive moment estimation algorithm, or *Adam* (Kingma & Ba, 2014), which is commonly used to train deep neural networks (Jais et al., 2019; Ogundokun et al., 2022). Here we provide the update rule used in Adam, but for an in-depth description of the ideas behind the algorithm, we refer to the original paper by Kingma and Ba (2014). At time steps $t = 1, 2, \ldots$, we follow the update rule

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon},$$

where η is the learning rate and ϵ is a small parameter added to avoid division by zero. All operations on vectors are element-wise. The two vectors $\hat{\boldsymbol{m}}_t$ and $\hat{\boldsymbol{v}}_t$ are unbiased estimates of the first and second moments of the gradient

$$\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}_{t-1}} J,$$

that is,

$$\boldsymbol{m}_t^* = E\big[\boldsymbol{g}_t\big] \quad \text{and} \quad \boldsymbol{v}_t^* = E\big[\boldsymbol{g}_t \odot \boldsymbol{g}_t\big].$$

They are updated in each iteration based on

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\, \boldsymbol{g}_{t-1}, \qquad \hat{\boldsymbol{m}}_t = \frac{\boldsymbol{m}_t}{1 - \beta_1^t},$$

and

$$\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\, \boldsymbol{g}_{t-1}^2, \qquad \hat{\boldsymbol{v}}_t = \frac{\boldsymbol{v}_t}{1 - \beta_2^t}.$$

The initial guesses $\boldsymbol{m}_0$ and $\boldsymbol{v}_0$ are zero vectors, and the parameters β_1 and β_2 are constants. We use the values recommended by the authors, that is, $\epsilon = 10^{-8}$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$ (Kingma & Ba, 2014). The learning rate η must be tuned to some value that gives stable convergence of the optimization problem. We cover this at the end of the next section.

3.2 Supervised Learning

Now that we have covered the structure of CNNs and discussed how the parameters $\boldsymbol{\theta}$ are updated, we switch our focus to the first approach to training neural networks: supervised learning. We start by explaining the two components of training that we left open in the previous section: data and objective functions. Similarly to the learning rate η used in the Adam optimizer, data and objective functions rely on hyperparameters, which must be tuned to ensure stable convergence and avoid overfitting. Therefore, we end this section with an overview of Bayesian optimization for hyperparameter tuning.

3.2.1 Data

In the supervised learning setting, we assume that we already have a data set containing N pairs of input states and target outputs. Let

$$\mathcal{D} = \{(s_i, f^*(s_i))\}_{i=1}^{N}$$

denote the data set, where $f^*(s_i)$ is the target output. This is a single action or a scalar value, depending on whether we train a policy or a value network, respectively.

Data are typically split into training, validation, and test sets. Training data $\mathcal{D}_{\text{train}}$ is used to tune the network parameters, validation data $\mathcal{D}_{\text{val}}$ is used to monitor the performance of the model during training, and test data $\mathcal{D}_{\text{test}}$ is used to evaluate the final model.

During training, the training data are randomly split into batches of some predetermined size n_{batch}. As in the previous section, we let

$$B_{\text{train}} = \{(s_i, f^*(s_i))\}_{i=1}^{n_{\text{batch}}}$$

denote one such batch. Each batch is fed into the network, and the objective function is calculated based on the n_{batch} network outputs. After all batches have been used to update the parameters, as discussed in Section 3.1.2, one epoch is completed. We repeat this process with new randomly sampled batches from the same training data for a predetermined number of epochs n_{epoch}.

The batch size must be appropriately tuned to extract as much information from the training data as possible, while avoiding overfitting. Overfitting occurs when the network learns patterns that are specific only to the training data and not the objective function, which may cause the training loss to be low, while the validation loss is high. We cover how this parameter is tuned in Section 3.2.4.

3.2.2 Objective Functions

Objective functions, denoted $J(\boldsymbol{\theta}; B_{\text{train}})$, are used to update the parameters in a neural network. In the supervised learning setting, they typically consist of two terms: a loss function and a regularization term. The loss function, denoted

$$\mathcal{L} = \mathcal{L}(\boldsymbol{\theta}; B_{\text{train}}),$$

is a measure of how much the network outputs $f(s_i; \boldsymbol{\theta})$ deviate from the target outputs $f^*(s_i)$, $i = 1, \ldots, n_{\text{batch}}$, based on a batch of data, B_{train}. The regularization term, denoted

$$\mathcal{R} = \mathcal{R}(\boldsymbol{\theta}),$$

is used to penalize large parameter values in order to avoid overfitting (Goodfellow et al., 2016). We now cover reasonable loss functions and a regularization term for policy and value networks.

A policy network returns a probability distribution over C possible actions, and consequently, a reasonable loss function is the cross-entropy metric (Cover & Thomas, 2005). For some target action a_i^*, let $\boldsymbol{p}_i^* \in \{0,1\}^C$ denote the target probability distribution, which has probability 1 on the index c_i^* corresponding to the target action. Then, given an estimated probability distribution $\boldsymbol{p}_i$, the cross-entropy of the two distributions is defined as

$$H(\boldsymbol{p}_i, \boldsymbol{p}_i^*) = -\sum_{c=1}^{C} p_{i,c}^* \log p_{i,c} = -\log p_{i,c_i^*}.$$

The lower the estimated probability of the correct action, the higher the cross-entropy. Then, for a batch of estimated distributions

$$(f(s_1;\boldsymbol{\theta}), \ldots, f(s_{n_{\text{batch}}};\boldsymbol{\theta}))$$

and target actions with associated distributions

$$(\boldsymbol{p}_1^*, \ldots, \boldsymbol{p}_{n_{\text{batch}}}^*),$$

the cross-entropy loss function is defined as

$$\mathcal{L}_{\text{CE}}(\boldsymbol{\theta}; B_{\text{train}}) = \frac{1}{n_{\text{batch}}} \sum_{i=1}^{n_{\text{batch}}} H(f(s_i;\boldsymbol{\theta}), \boldsymbol{p}_i^*). \tag{3.7}$$

A value network outputs a single scalar value, thus a reasonable loss function is the mean squared error (MSE) over all n_{batch} instances in the batch B_{train},

$$\mathcal{L}_{\text{MSE}}(\boldsymbol{\theta}; B_{\text{train}}) = \frac{1}{n_{\text{batch}}} \sum_{i=1}^{n_{\text{batch}}} (f(s_i;\boldsymbol{\theta}) - f^*(s_i))^2. \tag{3.8}$$

The regularization term used for both policy and value networks is proportional to the sum of squares of all parameters,

$$\mathcal{R}(\boldsymbol{\theta}) = \beta \|\boldsymbol{\theta}\|^2, \tag{3.9}$$

where the proportionality factor β is called *weight decay*. Similarly to n_{batch} and η, this is a hyperparameter that should be appropriately tuned.

Adding the respective loss functions and regularization terms together, given a batch of training data B_{train}, appropriate objective functions for training policy and value networks are given by

$$J_\pi(\boldsymbol{\theta}; B_{\text{train}}) = \mathcal{L}_{\text{CE}}(\boldsymbol{\theta}; B_{\text{train}}) + \mathcal{R}(\boldsymbol{\theta}) \tag{3.10}$$

and

$$J_v(\boldsymbol{\theta}; B_{\text{train}}) = \mathcal{L}_{\text{MSE}}(\boldsymbol{\theta}; B_{\text{train}}) + \mathcal{R}(\boldsymbol{\theta}), \tag{3.11}$$

respectively, with $\mathcal{L}_{\text{CE}}$, $\mathcal{L}_{\text{MSE}}$ and $\mathcal{R}$ defined in (3.7), (3.8) and (3.9).

3.2.3 Evaluation Metrics

The objective functions introduced in the previous section have the purpose of guiding the network training process, balancing between choosing the $\boldsymbol{\theta}$ that perfectly fits the training data and avoiding overfitting. Evaluation metrics are also used to measure deviation between network outputs and target outputs, but their purpose is to evaluate the networks on validation and test data, typically without regularization terms. In this subsection, we introduce some useful evaluation metrics, starting with those used for policy networks.

It is common to use two types of evaluation metrics for policy networks: one that is comparable to the training loss, to monitor how training and validation loss change compared to each other over time, and one that is more interpretable with regards to the purpose of the network. For the first case, it is reasonable to use cross-entropy validation loss, as defined in (3.7). This is calculated based on batches of data B_{val} for the sake of computational efficiency.

As a second and more interpretable evaluation metric for policy networks, Top-k accuracy is a common choice (Russakovsky et al., 2015). Let $c_i^{(k)}$ denote the index of the k-th largest probability in $f(s_i;\boldsymbol{\theta})$, and recall that c_i^* is the index associated with the target action a_i^*. Then, the Top-k metric based on a batch of data B_{val} is defined as

$$\text{Top}_k(\boldsymbol{\theta}; B_{\text{val}}) = \frac{1}{n_{\text{batch}}} \sum_{i=1}^{n_{\text{batch}}} \mathbb{I}\left(c_i^* \in \left\{c_i^{(j)}\right\}_{j=1}^{k}\right), \tag{3.12}$$

that is, the proportion of samples for which the correct action is among the k actions assigned the highest probability by the network. When monitoring the accuracy of policy networks, we typically choose a couple values of k, say 1 and 3, and monitor Top-1 and Top-3 accuracy. For value networks, MSE as defined in (3.8) is typically used for both training and validation, as it already provides an intuitive metric for network performance.

3.2.4 Hyperparameter Tuning with Bayesian Optimization

In the previous subsections, we introduced three network hyperparameters that must be tuned: batch size n_{batch}, learning rate η, and weight decay β. Let these be denoted by

$$\boldsymbol{\xi} = (n_{\text{batch}}, \eta, \beta).$$

Given some validation loss $\mathcal{L}_{\text{val}}$, which is typically chosen based on the ones defined in the previous subsections, the purpose of hyperparameter tuning is to find the $\boldsymbol{\xi}$ that minimizes the validation loss,

$$\boldsymbol{\xi}^* = \arg\min_{\boldsymbol{\xi}\in\Xi} \mathcal{L}_{\text{val}}(\boldsymbol{\theta}(\boldsymbol{\xi}); B_{\text{val}}),$$

which is an optimization problem over a predetermined parameter space Ξ. Here, we treat $\boldsymbol{\theta}$ as a function of the hyperparameters $\boldsymbol{\xi}$. For ease of notation, we write

$$\mathcal{L}_{\text{val}}(\boldsymbol{\xi}) \equiv \mathcal{L}_{\text{val}}(\boldsymbol{\theta}(\boldsymbol{\xi}); B_{\text{val}}).$$

Each evaluation of $\mathcal{L}_{\text{val}}(\boldsymbol{\xi})$ requires training an entire neural network, which is a very expensive procedure. To handle optimization problems with expensive objective functions, Bayesian optimization techniques have proven successful (Brochu et al., 2010).

Bayesian optimization techniques rely on two components: a *surrogate model*, which for each $\boldsymbol{\xi}$ gives an estimate $\hat{\mathcal{L}}_{\text{val}}(\boldsymbol{\xi})$ and a corresponding uncertainty of that estimate,

and an *acquisition function*, which is used to determine what $\boldsymbol{\xi}$ to try next, based on the surrogate model. Generally speaking, the idea is to balance exploiting hyperparameters that have previously given the lowest observed values of $\mathcal{L}_{\text{val}}$ and exploring parameters that are assigned high uncertainty by the surrogate model.

A common surrogate model is the Tree-structured Parzen Estimator (TPE) (Bergstra et al., 2011). TPE uses the previous observations

$$\mathcal{L}_{\text{val}}(\boldsymbol{\xi}_q), \quad q = 1, \ldots, t-1,$$

to form two densities,

$$l(\boldsymbol{\xi}) \;=\; p\big(\boldsymbol{\xi} \mid \mathcal{L}_{\text{val}}(\boldsymbol{\xi}) < y^*\big) \quad \text{and} \quad g(\boldsymbol{\xi}) \;=\; p\big(\boldsymbol{\xi} \mid \mathcal{L}_{\text{val}}(\boldsymbol{\xi}) \geq y^*\big),$$

where y^* is the 0.15-quantile of the observed losses (Bergstra et al., 2011). That is, $l(\boldsymbol{\xi})$ and $g(\boldsymbol{\xi})$ are estimated so that they fit the 15% best and the 85% worst of all previous observations, respectively. The two densities are estimated using Parzen window density estimators. For details on how this is done, we refer to the original paper by Bergstra et al. (2011).

Once we have estimated $l(\boldsymbol{\xi})$ and $g(\boldsymbol{\xi})$, we determine the set of parameters $\boldsymbol{\xi}_t$ to try next. For this purpose, the TPE surrogate model naturally combines with the expected improvement (EI) acquisition function, defined as

$$\text{EI}(\boldsymbol{\xi}) = E[\max\{0, y^* - \hat{\mathcal{L}}_{\text{val}}(\boldsymbol{\xi})\}]. \tag{3.13}$$

Maximizing (3.13) is equivalent to maximizing the ratio between $l(\boldsymbol{\xi})$ and $g(\boldsymbol{\xi})$ (Snoek et al., 2012), meaning that we choose the next set of parameters $\boldsymbol{\xi}_t$ that maximizes the EI acquisition function by choosing

$$\boldsymbol{\xi}_t = \arg\max_{\boldsymbol{\xi} \in \Xi} \frac{l(\boldsymbol{\xi})}{g(\boldsymbol{\xi})}. \tag{3.14}$$

Intuitively, this means that we choose the $\boldsymbol{\xi}$ that is likely to be good and unlikely to be bad. In practice, maximization is performed by sampling several $\boldsymbol{\xi}$ from $l(\boldsymbol{\xi})$, and then choose according to (3.14).

The optimization algorithm is run for some predetermined number of trials, r, after which we choose

$$\hat{\boldsymbol{\xi}}^* = \arg\min_{q=1,\ldots,r} \mathcal{L}_{\text{val}}(\boldsymbol{\xi}_q)$$

among the r sets of parameters we have evaluated.

Not all parameters tested in hyperparameter tuning contribute equally to reducing the validation loss. A common way to measure *hyperparameter importance* is through the use of permutation importance (Altmann et al., 2010). Here, we provide a brief overview of this method. For in-depth explanations of random forests and the use of permutation importance, we refer to the literature by Breiman (2001) and Altmann et al. (2010), respectively. When calculating permutation importance, we first fit a random forest regressor $\hat{f}$ to the data set

$$\{\boldsymbol{\xi}_q, \mathcal{L}_{\text{val}}(\boldsymbol{\xi}_q)\}_{q=1}^{r},$$

that takes a vector of hyperparameters $\boldsymbol{\xi}_q$ as input and predicts the associated validation loss, $\hat{f}(\boldsymbol{\xi}_q)$. Next, we shuffle the r instances of one of the hyperparameters, say hyperparameter j, thus obtaining $\boldsymbol{\xi}_{q,\text{perm}(j)}$, and predict the resulting validation loss using the

random forest on the permutated data, $\hat{f}(\boldsymbol{\xi}_{q,\text{perm}(j)})$, $q = 1, \ldots, r$. Based on this, we calculate the MSE between the predictions when permuting hyperparameter j, and the original validation loss,

$$\text{MSE}_{\text{perm}(j)} = \frac{1}{r}\sum_{q=1}^{r}\big(\hat{f}(\boldsymbol{\xi}_{q,\text{perm}(j)}) - \mathcal{L}_{\text{val}}(\boldsymbol{\xi}_q)\big)^2.$$

By repeating this process several times, calculating the mean MSE from permutations of each hyperparameter, we obtain measures of how much the permutation of hyperparameter j changes the prediction of the random forest. The average $\text{MSE}_{\text{perm}(j)}$ is therefore a measure of how important each hyperparameter is to the change in the validation loss. Then, the permutation importance of hyperparameter j is defined as

$$I_j = \overline{\text{MSE}}_{\text{perm}(j)} - \text{MSE}_{\text{orig}}, \tag{3.15}$$

where the first term is the mean MSE over all the permutations of hyperparameter j, and MSE_{orig} is based on the prediction on the original hyperparameter set,

$$\text{MSE}_{\text{orig}} = \frac{1}{r}\sum_{q=1}^{r}\big(\hat{f}(\boldsymbol{\xi}_q) - \mathcal{L}_{\text{val}}(\boldsymbol{\xi}_q)\big)^2.$$

3.3 Proximal Policy Optimization

In this section, we cover the second machine learning approach that we use to approximate π^* and v^*: PPO (Schulman et al., 2017). PPO is a state-of-the-art reinforcement learning technique that has proven successful in solving a variety of sequential decision-making problems, including games (Berner et al., 2019). It builds on the core principles of reinforcement learning, where an agent iteratively interacts with an environment, earning rewards for its actions and gradually learning to improve its policy based on the reward signals. This concept is illustrated in Figure 3.2. In our case, the environment is the *Former* game, and PPO is a way of training an agent to play the game through trial and error.

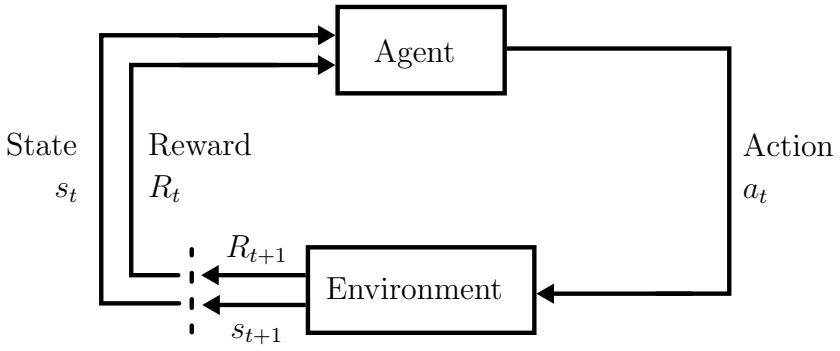


Figure 3.2: The agent-environment interaction of reinforcement learning. An agent performs an action a_t at time t, which changes the state of the environment from s_t to s_{t+1} and gives a reward R_{t+1}. The agent then repeats the process.

We start by providing some background theory on how reinforcement learning algorithms find optimal policy and value functions through the use of dynamic programming techniques, specifically through *policy iteration*. Then, we explain how PPO approximates policy iteration, thus finding approximations to the optimal policy and value functions.

3.3.1 Dynamic Programming

Within the notation and objectives defined by the MDP framework in Section 2.3.1, dynamic programming algorithms are used to find optimal policy and value functions (Sutton & Barto, 2014). This is done through a process called policy iteration, which consists of two steps: policy evaluation, where we use the current policy π to find a corresponding value function v_π, and policy improvement, where we use the new v_π to improve the previous policy. In this subsection, we explain the details of the two steps in policy iteration, starting with policy evaluation.

During policy evaluation, we assume that the agent acts according to some fixed policy π. The purpose is to find a new value function, v_π, based on this policy. Generalizing the definition of the optimal value function in Section 2.3.1, any value function $v_\pi(s)$ is defined as the expected return from some state when acting according to π,

$$v_\pi(s) = E_\pi[G_0 \mid s]. \tag{3.16}$$

Here, using our definition of future return in (2.1), where we recall that we set the discounting factor $\gamma = 1$, (3.16) can be written as the immediate reward from the next action $R(s, a)$ and the return from the next state G_1,

$$v_\pi(s) = E_\pi[R(s, a) + G_1 \mid s].$$

Applying the law of total expectation gives

$$v_\pi(s) = \sum_{a \in \mathcal{A}_s} \pi(a \mid s)\, E_\pi\left[R(s, a) + G_1 \mid s, a\right],$$

and since $R(s, a)$ is deterministic once s and a are fixed,

$$v_\pi(s) = \sum_{a \in \mathcal{A}_s} \pi(a \mid s)\left(R(s, a) + E_\pi\left[G_1 \mid s, a\right]\right).$$

Then, writing out the expectation, we have

$$v_\pi(s) = \sum_{a \in \mathcal{A}_s} \pi(a \mid s) \left(R(s, a) + \sum_{s' \in \mathcal{S}} P(s' \mid s, a)\, E_\pi\left[G_1 \mid s, a, s'\right] \right).$$

Since the future return G_1 only depends on the future state s', the last expectation is in fact the value function evaluated in the next state, thus we end up with the *Bellman expectation equation*,

$$v_\pi(s) = \sum_{a \in \mathcal{A}_s} \pi(a \mid s) \left(R(s, a) + \sum_{s' \in \mathcal{S}} P(s' \mid s, a)\, v_\pi(s') \right). \tag{3.17}$$

This is the core component in policy evaluation. It is used to define the policy evaluation scheme, where in time steps $k = 1, 2, \ldots,$

$$v^{(k+1)}(s) = \sum_{a \in \mathcal{A}_s} \pi(a \mid s) \left(R(s, a) + \sum_{s' \in \mathcal{S}} P(s' \mid s, a)\, v^{(k)}(s') \right), \tag{3.18}$$

where $v^{(0)}$ can be defined arbitrarily. Note that, by construction, v_π is a fixed point in this iteration. Moreover, the iteration scheme is guaranteed to converge to v_π for a fixed policy π, assuming the MDP is finite (Sutton & Barto, 2014). This is the case for *Former*.

In policy improvement, we keep v_π fixed and update π. The idea is to look one move ahead and update the policy so that it assigns probability 1 to the action that gives the highest expected reward after that move. Mathematically, we define the action-value function,

$$q_\pi(s,a) = R(s,a) + \sum_{s' \in \mathcal{S}} P(s' \mid s,a)\, v_\pi(s'),$$

which is the expected return from taking action a in state s, and then following the policy π. Note that this is the same expression as in the parentheses of (3.17), hence there is a close relationship between q_π and v_π. Based on the action-value function, we define the updated policy function to be

$$\pi_{\text{new}}(a \mid s) = \begin{cases} 1, & a = \arg\max\limits_{a' \in \mathcal{A}_s} q_\pi(s,a'), \\ 0, & \text{otherwise.} \end{cases}$$

By the policy improvement theorem (Sutton & Barto, 2014), this step is guaranteed to provide a new policy π_{new} that is better than or equal to the previous policy, in the sense that

$$v_{\pi_{\text{new}}}(s) \geq v_\pi(s) \quad \forall s \in \mathcal{S}.$$

Moreover, by repeating policy evaluation and policy improvement over and over, the policy and value functions will converge to their respective optimal functions.

Recall from Section 2.3.2 that the problem of solving *Former* is formulated as finding π^* and v^*. This is the purpose of policy iteration, hence it is a reasonable approach to solving our problem. However, as we also discussed in Section 2.3.2, it is in practice impossible to find π^* and v^* since it involves visiting every $s \in \mathcal{S}$. Therefore, we approximate the two-step procedure using PPO. This is the topic of the next subsection.

3.3.2 PPO as an Approximate Policy Iteration

The purpose of PPO is to train a neural network, which is done through policy iteration. The method uses a *dual-head* CNN, which is a type of neural network consisting of several convolutional blocks that are connected in the last layer to two *heads*: one that approximates the optimal policy function, denoted $\pi_{\boldsymbol{\theta}}$, and one that approximates the optimal value function, denoted $v_{\boldsymbol{\phi}}$. Here, $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ are the parameters in the respective networks. This is a typical setup in *actor-critic* reinforcement learning models, where the policy head (the actor) chooses the action, and the value network (the critic) evaluates the action chosen by the actor (Mnih et al., 2016). PPO trains the dual-head neural network by approximating the policy iteration procedure, repeatedly updating $v_{\boldsymbol{\phi}}$ and $\pi_{\boldsymbol{\theta}}$ through policy evaluation and policy improvement, respectively. We now cover how both are done, starting with policy evaluation.

Approximate Policy Evaluation

As explained in the previous subsection, the purpose of policy evaluation is to update $v_{\boldsymbol{\phi}}$ while keeping $\pi_{\boldsymbol{\theta}}$ fixed. In PPO, this is performed by tuning the parameters $\boldsymbol{\phi}$, that is, by

training the critic. As discussed in Section 3.1.2, this requires data, an objective function, and an optimization algorithm.

The PPO agent collects a batch of data of size n_{batch} through repeated interactions with the environment, which in our case translates to the agent playing many games of *Former*. If the duration of a game is T moves, the agent obtains T samples

$$\{(s_t, a_t, R_t)\}_{t=0}^{T-1},$$

where s_t is the state, a_t is the action chosen, and the R_t is the reward obtained at each time step in that game. For each sample, we then calculate the temporal difference,

$$\hat{R}_t = \sum_{h=0}^{T-1} \lambda^h \, R_{t+h} \; + \; \lambda^T \, v_\phi(s_{t+T}). \tag{3.19}$$

The temporal difference is a biased estimate of the rewards remaining in an episode from a state s_t, under the current actor, $\pi_{\boldsymbol{\theta}}$. The bias comes from the temporal difference parameter $\lambda \in [0, 1]$, which is incorporated to reduce the high variance associated with full Monte Carlo samples. Konda and Tsitsiklis (1999) showed that introducing this bias often stabilizes convergence, and in practice it is common to set $\lambda \in [0.9, 0.99]$ (Schulman et al., 2015; Sutton & Barto, 2014).

The agent obtains a dataset $\mathcal{D}_{\text{train}}$ containing N samples by playing many episodes after each other. As explained in Section 3.1.2, the data is split into batches of size n_{batch}, denoted B_{train}. From each temporal difference in a batch of data, we find the advantage estimates

$$A_i = \hat{R}_i - v_{\boldsymbol{\phi}}(s_i), \qquad i = 1, \ldots, n_{\text{batch}}.$$

These form the basis of the loss function associated with policy evaluation with PPO. Given a batch of data, the *critic loss* is calculated using

$$\mathcal{L}_{\text{V}}(\boldsymbol{\phi}; B_{\text{train}}) = \frac{1}{n_{\text{batch}}} \sum_{i=1}^{n_{\text{batch}}} \big(\hat{R}_i - v_{\boldsymbol{\phi}}(s_i)\big)^2, \tag{3.20}$$

that is, the MSE between the critic prediction $v_{\boldsymbol{\phi}}(s_i)$ and the temporal difference $\hat{R}_i$ of each sample. Since $\hat{R}_i$ estimates the remaining rewards from a state, it is essentially a sample-based estimate of the Bellman expectation defined in (3.17). Consequently, by minimizing (3.20), PPO approximates the fixed point solution to the policy evaluation scheme (3.18) each time the parameters $\boldsymbol{\phi}$ are updated.

Approximate Policy Improvement

Recall that, in the policy improvement step, the value function is fixed and used to update the policy function. Equivalently, in the policy improvement step of PPO, the value head $v_{\boldsymbol{\phi}}$ is fixed, and we train the policy head $\pi_{\boldsymbol{\theta}}$. For a detailed explanation of the ideas behind the PPO policy improvement step, we refer to the original paper by Schulman et al. (2017). Here, we provide a brief overview, without a detailed explanation of the ideology. First, define the probability ratio between the new and old policies for each sample i,

$$r_i(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(a_i \mid s_i)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a_i \mid s_i)},$$

where, as in Section 2.3.2, $\pi_{\boldsymbol{\theta}}(a \mid s)$ denotes the probability assigned to action a by $\pi_{\boldsymbol{\theta}}(s)$. With this definition, and based on the same samples of data obtained as explained under policy evaluation, the clipped surrogate objective function used by PPO is defined as

$$\mathcal{L}_{\text{CLIP}}(\boldsymbol{\theta}; B_{\text{train}}) = -\frac{1}{n_{\text{batch}}} \sum_{i=1}^{n_{\text{batch}}} \min\left(r_i(\boldsymbol{\theta})\, A_i, \text{clip}(r_i(\boldsymbol{\theta}), 1-\varepsilon, 1+\varepsilon)\, A_i\right). \tag{3.21}$$

This is typically referred to as the *actor loss*. In (3.21), the clip$(\cdot)$ function is defined as

$$\text{clip}(r_i(\boldsymbol{\theta}),\, 1-\varepsilon,\, 1+\varepsilon) = \max\Big(\min\big(r_i(\boldsymbol{\theta}),\, 1+\varepsilon\big),\, 1-\varepsilon\Big),$$

which ensures that the policy does not undergo excessively large changes in each step (Schulman et al., 2017).

Combined Objective Function

In practice, a step in the policy iteration using PPO is carried out in a single step, and not by alternating between policy evaluation and improvement. This is done by combining the actor loss (3.21) and the critic loss (3.20) into one. On top of this, it is common to subtract an entropy bonus term (Schulman et al., 2017),

$$\mathcal{H}(\boldsymbol{\theta}; B_{\text{train}}) = -\frac{1}{n_{\text{batch}}} \sum_{i=1}^{n_{\text{batch}}} \sum_{a \in \mathcal{A}_{s_i}} \pi_{\boldsymbol{\theta}}(a \mid s_i)\, \log \pi_{\boldsymbol{\theta}}(a \mid s_i), \tag{3.22}$$

to motivate exploration: deterministic probability distributions give low entropy bonus, and vice versa. Combining the three, the objective function that PPO minimizes for each batch of data B_{batch} collected is given by

$$J(\boldsymbol{\theta}, \boldsymbol{\phi}; B_{\text{train}}) = \underbrace{\mathcal{L}_{\text{CLIP}}(\boldsymbol{\theta}; B_{\text{train}})}_{\text{actor loss}} + c_{\text{val}} \underbrace{\mathcal{L}_{\text{V}}(\boldsymbol{\phi}; B_{\text{train}})}_{\text{critic loss}} - c_{\text{ent}} \underbrace{\mathcal{H}(\boldsymbol{\theta}; B_{\text{train}})}_{\text{entropy}}, \tag{3.23}$$

where c_{val} and c_{ent} are constants, and $\mathcal{L}_{\text{CLIP}}$, $\mathcal{L}_{\text{V}}$ and $\mathcal{H}$ are defined in (3.21), (3.20) and (3.22), respectively. For the task of minimizing (3.23), it is common to use the Adam optimizer, which we discussed in Section 3.1.2.

CHAPTER

FOUR

SEARCH TECHNIQUES

Search techniques are clever ways to look for solutions to sequential decision-making problems. They have in particular proven useful for solving deterministic single-player games, where a solution always exists, but the problem is finding it in a large state space (Pearl & Korf, 1987; Schadd et al., 2008). In this chapter, we first formulate *Former* as a tree search problem and explain how we use the approximate policy and value functions (π and v) to efficiently traverse the search tree (Section 4.1). Thereafter, we explain the two search techniques we use in this thesis: MCTS (Section 4.2) and beam search (Section 4.3).

4.1 *Former* as a Tree Search Problem

In a deterministic, single-player game such as *Former*, it is natural to model boards and actions as a search tree. Each node then represents a game state, and edges correspond to actions. The root node is the initial state, and nodes reachable in t actions lie at tree level t. Applying an action to a state generates a *child* node, which is connected to its *parent* node by the edge corresponding to the action taken. In Figure 4.1 we illustrate how such a tree structure represents states and actions for a simple 2×2 board with 2 different shapes.

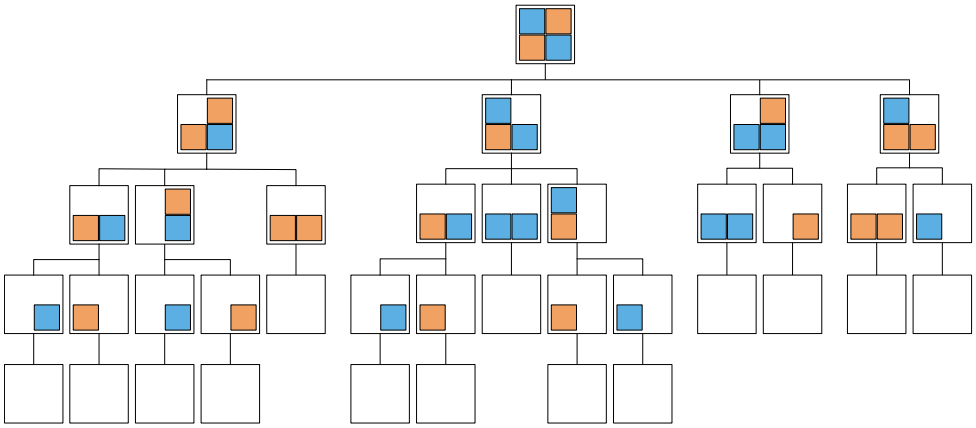


Figure 4.1: A simple initial 2×2 board and the corresponding search tree. Nodes are the boards, and edges represent each possible action.

As we observed in Section 2.2.2, the number of possible combinations of actions explodes as the number of moves increases, reaching over 10^{21} possible combinations at a depth of 15 (for 9 by 7 boards with 4 shapes). This implies that a solution cannot be found by exploring the entire search space, and consequently we need to narrow it down by choosing actions wisely at each level. This is where the policy and value functions, π and v, come into play. Policy functions indicate what actions to take from one board, whereas value functions can be used to determine what actions are the most promising from an entire tree level by comparing board difficulty after each action. These two are incorporated into the search techniques that we use: π in MCTS, and v in beam search.

4.2 Monte Carlo Tree Search

MCTS is a search method that has proven particularly useful for solving sequential decision-making problems with large state spaces (Świechowski et al., 2023). The general idea behind the method is to estimate the value of each possible action from a state based on Monte Carlo simulations, and use these values to determine the best choice of action. Kocsis and Szepesvári (2006) and Coulom (2006) first proposed the algorithm, using it to significantly improve machine-player performance in *Go*. MCTS has since been used to solve a variety of sequential decision-making problems, while also being improved by incorporating techniques from other areas, such as reinforcement learning. Arguably, the most well-known use cases are AlphaGo and its successor models AlphaGo Zero and AlphaZero, which combined MCTS with deep neural networks to outperform all other *Go* players, both humans and machines (Silver et al., 2016, 2017, 2018).

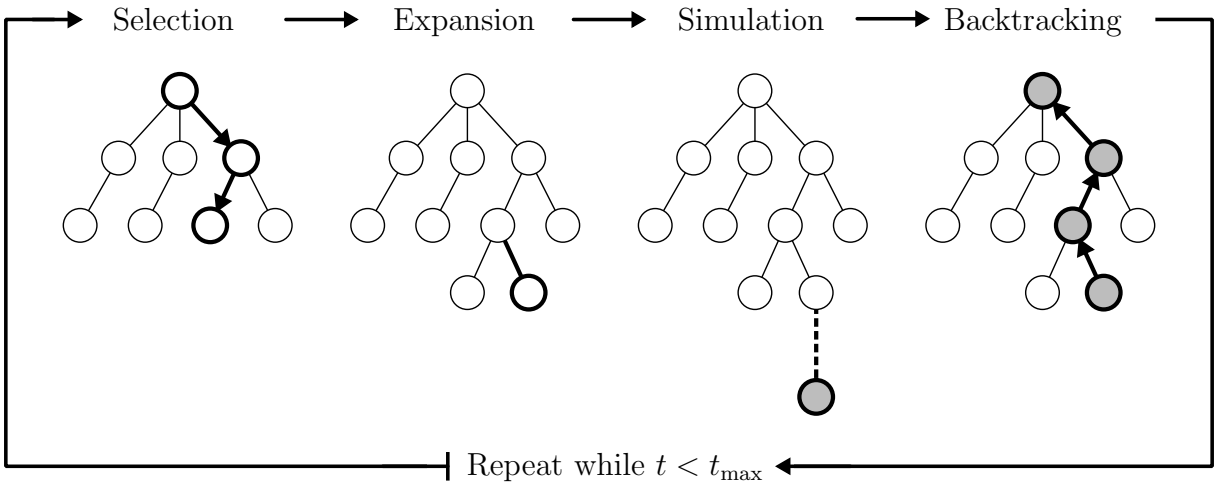


Figure 4.2: The four steps of the MCTS algorithm. In step 1 we traverse the existing tree, in step 2 we add all child nodes and choose one action, in step 3 we simulate a play-through from that action to obtain a value based on the outcome of the simulation, which is added to the traversed nodes in step 4. The 4 steps are repeated until some time limit $t_{\max}$ is reached.

Each iteration of the MCTS algorithm is split into four steps: selection, expansion, simulation and backtracking. These are illustrated in Figure 4.2. The purpose of the four steps is to estimate the value of each possible action from the initial state and use these estimates to choose the best action. In the first step, we select a node that represents a

new state in the game (selection). Then, we add all its child nodes to our existing tree (expansion), simulate a play-through of the game from one of these nodes (simulation), and store the value obtained from the play-through as a measure of quality of the selected tree branch (backtracking). Throughout this process, we keep track of the best solution found, which is returned at the end of the search, when some time limit $t_{\max}$ has been reached.

In this section, we explain the details of the MCTS algorithm, highlighting the adaptations we make to solve *Former*. An overview of the implementation is shown in Algorithm 1, and we explain each step thoroughly in separate subsections.

Algorithm 1 MCTS

Require: initial state s_0, time limit $t_{\max}$, policy function π
initialize tree with root s_0 ▷ Each iteration starts from s_0
$v_{\text{best}} \leftarrow \infty$
while $t < t_{\max}$ **do**
    $s_{\text{leaf}} \leftarrow$ SELECTION(s_0, π) ▷ see Algorithm 1.1
    $s_{\text{new}} \leftarrow$ EXPANSION(s_{leaf}, π) ▷ see Algorithm 1.2
    $v_{\text{sim}} \leftarrow$ SIMULATION(s_{new}, π) ▷ see Algorithm 1.3
    BACKTRACKING(v_{sim}, path from s_0 to s_{new}) ▷ see Algorithm 1.4
    if $v_{\text{sim}} < v_{\text{best}}$ **then**
        store actions taken from s_0 to s_T
        $v_{\text{best}} \leftarrow v_{\text{sim}}$
    end if
end while
return shortest sequence of actions from s_0 to s_T

Step 1: Selection

In the selection phase of MCTS, we traverse the tree from the root node until we reach a leaf node, as described in Algorithm 1.1. A leaf node is defined as a node that we have not yet expanded, that is, it has no children. We choose which node to traverse to at each tree level based on some decision rule, which we tune based on the exploration-exploitation trade-off: we want to explore actions that have not been thoroughly explored while also exploiting the current best-performing actions. A common decision rule for MCTS is to choose the action that maximizes the upper confidence bound for trees (UCT) formula (Silver et al., 2016),

$$Q(s,a) + \pi\left(a \mid s\right) c_{\text{puct}} \frac{\sqrt{\sum_{a' \in \mathcal{A}_s} N(s,a')}}{1 + N(s,a)}, \tag{4.1}$$

where $N(s,a)$ is the number of times action a has been visited from state s, c_{puct} is a tuning parameter, and $\pi(a \mid s)$ is the probability assigned to action a in state s by the policy. The function $Q : \mathcal{S} \times \mathcal{A}_s \to \mathbb{R}$ is the average simulation value obtained,

$$Q(s,a) = \frac{\sum_{i=1}^{N(s,a)} V_{\text{sim}}^{(i)}(s,a)}{N(s,a)}, \tag{4.2}$$

where $V_{\text{sim}}^{(i)}(s,a)$ is the value from the i-th simulation where action a was explored from state s during the MCTS iteration. We explain how this value is defined when we discuss the simulation step.

In the UCT formula (4.1), $Q(s,a)$ represents exploitation and the second term represents exploration: Q is higher the better the previous simulation values, and the second term is higher the fewer times a node has been visited. The trade-off is determined by the parameter c_{puct}, which is increased to obtain more exploration, and vice versa.

Algorithm 1.1 MCTS: Selection

Require: initial state s_0, policy function π

$s \leftarrow s_0$
while node representing s is not leaf node **do**
    $a^* \leftarrow \arg\max_a \left[Q(s,a) + c_{\text{puct}}\ \pi(a \mid s)\ \frac{\sqrt{\sum_{a' \in \mathcal{A}_s} N(s,a')}}{1+N(s,a)} \right]$
    $s \leftarrow \tau(s, a^*)$
end while
return $s_{\text{leaf}} \leftarrow$ node

Step 2: Expansion

When selection terminates at a leaf node s_{leaf}, we expand it by generating all untried actions as children, and simulate from one of them. The choice of child is made by sampling from the policy, $a \sim \pi(s_{\text{leaf}})$. When expanding, we also initialize the number of visits N, the average value Q, and the probability assigned to each child node by the policy. Pseudocode for the expansion step is shown in Algorithm 1.2.

Algorithm 1.2 MCTS: Expansion

Require: leaf node s_{leaf}, policy function $\pi(a \mid s)$

for all actions $a \in \mathcal{A}_{s_{\text{leaf}}}$ **do**
    $s' \leftarrow \tau(s_{\text{leaf}}, a)$
    add s' as child of s_{leaf}
    initialize $N(s_{\text{leaf}}, a) \leftarrow 0,\ Q(s_{\text{leaf}}, a) \leftarrow 0,\ \pi(a \mid s_{\text{leaf}})$
end for
$s_{\text{new}} \leftarrow \tau(s_{\text{leaf}}, a)$, where $a \sim \pi(s_{\text{leaf}})$
return s_{new}

Step 3: Simulation

After expanding the tree and choosing an initial action to simulate from, we simulate the rest of the game. The purpose of this is to obtain some value that indicates the quality of the chosen actions. We perform the simulation by sampling from the policy at each turn, as explained in the previous step.

When there are no more actions to be performed, the simulation step is completed and we obtain some value V_{sim} based on the simulation performance. The value must reflect the goal of *Former*, meaning that a low total number of moves should give a high value. Consequently, a reasonable definition of V_{sim} is

$$V_{\text{sim}} = -T,$$

that is, the negative number of moves used to clear the board. The simulation step implementation is shown in Algorithm 1.3.

Algorithm 1.3 MCTS: Simulation

Require: expanded node s_{new}, policy function π, current tree level d

$s \leftarrow s_{\text{new}}$
while $\mathcal{A}_s \neq \varnothing$ **do**
    sample $a \sim \pi(s)$
    $s \leftarrow \tau(s, a)$
    $d \leftarrow d + 1$ ▷ Keep track of number of moves used
end while
$T \leftarrow d$
$V_{\text{sim}} \leftarrow -T$ ▷ Negative number of moves used to clear the board
return V_{sim}

Step 4: Backtracking

After the simulation, we backtrack along the nodes visited in steps 1, 2, and 3, incrementing the visit count $N(s, a)$ and updating the mean value using

$$Q(s, a) \leftarrow \frac{(N(s, a) - 1)\, Q(s, a) + V_{\text{sim}}}{N(s, a)}.$$

The backtracking step is included in Algorithm 1.4.

Algorithm 1.4 MCTS: Backtracking

Require: simulation value V_{sim}, path from s_0 to s_{new}

for all (s, a) in path **do**
    $N(s, a) \leftarrow N(s, a) + 1$ ▷ Increment visit count
    $Q(s, a) \leftarrow \frac{(N(s, a) - 1)\, Q(s, a) + V_{\text{sim}}}{N(s, a)}$ ▷ Update mean simulation value
end for

We repeat the four steps of the MCTS algorithm until some time limit $t_{\max}$ is reached. Although this limit can be set arbitrarily large, our goal is to solve *Former* boards efficiently, and hence $t_{\max}$ is chosen to reflect that purpose.

Minimum Exploration Limit

Silver et al. (2016) and Schadd et al. (2008) showed that it is beneficial to set a minimum exploration limit from the first state, $N_{\min}$. This is due to the importance of the first few moves, where the MCTS model may hone in on just a select few actions very quickly if the parameter c_{puct} is improperly tuned. The minimum exploration limit forces the model to try each action from the initial state $N_{\min}$ times before the UCT formula (4.1) takes over.

4.3 Beam Search

Beam search (Bisiani, 1987) is conceptually a simpler search technique than MCTS. It does not rely on full simulations of the game, but rather plays a game one move at a time, choosing the w_{beam} best actions available according to some model at each tree level. There is no randomness or simulations involved, and as a result, the method relies more on correct guidance than MCTS. Given a good model to guide it, however, the method can find solutions to a *Former* board quickly, as was demonstrated by Odland (2024). In this section, we briefly explain how the algorithm works. Pseudocode is included in Algorithm 2.

Algorithm 2 Beam Search

Require: initial state s_0, beam width w_{beam}, value function $v(\cdot)$

$\mathcal{B} \leftarrow \{s_0\}$ ▷ current beam
while true **do**
  $\mathcal{C} \leftarrow \varnothing$ ▷ collect all child states
  for all $s \in \mathcal{B}$ **do**
    for all $a \in \mathcal{A}_s$ **do**
      $s' \leftarrow \tau(a, s)$
      $\mathcal{C} \leftarrow \mathcal{C} \cup \{s'\}$
    end for
  end for
  Compute value $v(s')$ for each $s' \in \mathcal{C}$
  Sort $\mathcal{C}$ in descending order by v
  $\mathcal{B} \leftarrow \{\text{first } w_{\text{beam}} \text{ states in } \mathcal{C}\}$ ▷ Prioritize child states using v
  if any $s \in \mathcal{B}$ is a goal state **then**
    return solution path ▷ Terminate once first solution is found
  end if
end while

Beam search only explores a fixed number of actions at each tree level. Consequently, it builds a tree that reminds of a "beam", with a fixed maximum width w_{beam}. Starting from the root, we expand all child nodes from the previous layer, score each new state using some value function $v(s)$, and keep only the top w_{beam} states for the next level. Figure 4.3 illustrates how beam search prioritizes actions based on a heuristic and builds a tree of fixed width $w_{\text{beam}} = 2$.

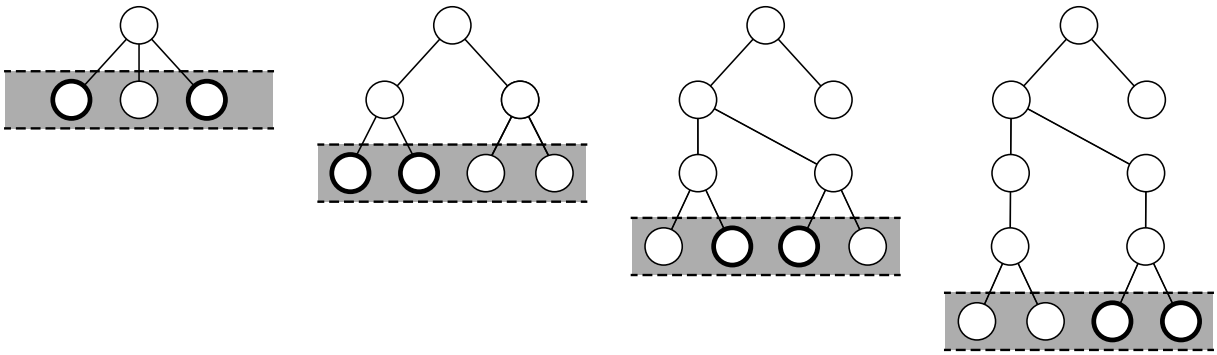


Figure 4.3: Illustrative example of beam search. At each tree level, the $w_{\text{beam}} = 2$ best actions according to some heuristic are added to the tree. This causes the tree to have a fixed width, which reduces runtime and space-complexity.

CHAPTER

FIVE

METHODOLOGY

Now that we have developed the theoretical foundations, we show how the theory is implemented in practice. Recall from the end of Chapter 2 that our approach can be roughly split into two steps:

1. Make *models* that approximate $\pi^*(s)$ and $v^*(s)$, whose purposes are to suggest actions and predict the number of remaining moves from state s.

2. Make *solvers* by combining the models with MCTS and beam search.

We use three approaches to create models: self-made heuristics, supervised learning, and PPO. These are not meant to solve the game on their own, but merely suggest promising actions or predict the difficulty of some board. Models are combined with search techniques to form solvers, which are used to search for the optimal solutions to any board. Here, policy models are combined with MCTS and value models are combined with beam search. An overview of the two main steps, the five subcomponents, and how they form a pipeline for solving *Former* is illustrated in Figure 5.1.

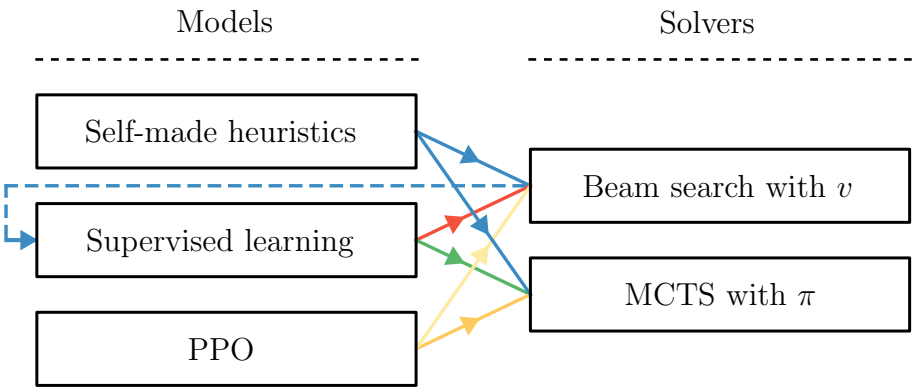


Figure 5.1: The methods we use to solve *Former* and their relations. Self-made heuristics, supervised learning and PPO are used independently to make models, which approximate π^* and v^*. These are combined into MCTS and beam search to form solvers, which are used to search for solutions to *Former* boards. Self-made heuristics are also used with beam search to generate data for the supervised learning approach.

This chapter provides details of the implementation of each component of our approach. We first give a brief note on how *Former* is implemented and how we acquire daily

boards in Section 5.1. Then, we proceed with each of the three model types, separately. First, we explain our choice of self-made heuristics in Section 5.2. Second, Section 5.3 gives the details of the supervised learning methodology, and third, Section 5.4 provides the implementation details of PPO. Finally, in Section 5.5, we switch our focus to the solvers, explaining the implementation details of MCTS and beam search as well as how the three models are incorporated into them.

5.1 Code Implementation and Daily Board Acquisition

Former is not an open-source game, so we implement the game ourselves. Due to the computationally heavy methods we use in this thesis, fast function evaluations are important. Therefore, we implement the game in C++. For comparison, the empirical cumulative distribution functions (CDFs) of the time it takes to solve *Former* boards with random play using Python and C++ are shown in Figure 5.2. The Python implementation had a median runtime of 0.0021 seconds, while C++ used 0.00027 seconds: approximately 8 times faster. These results, and all subsequent runtimes reported in this thesis, were recorded on a MacBook Air with an Apple M1 chip.

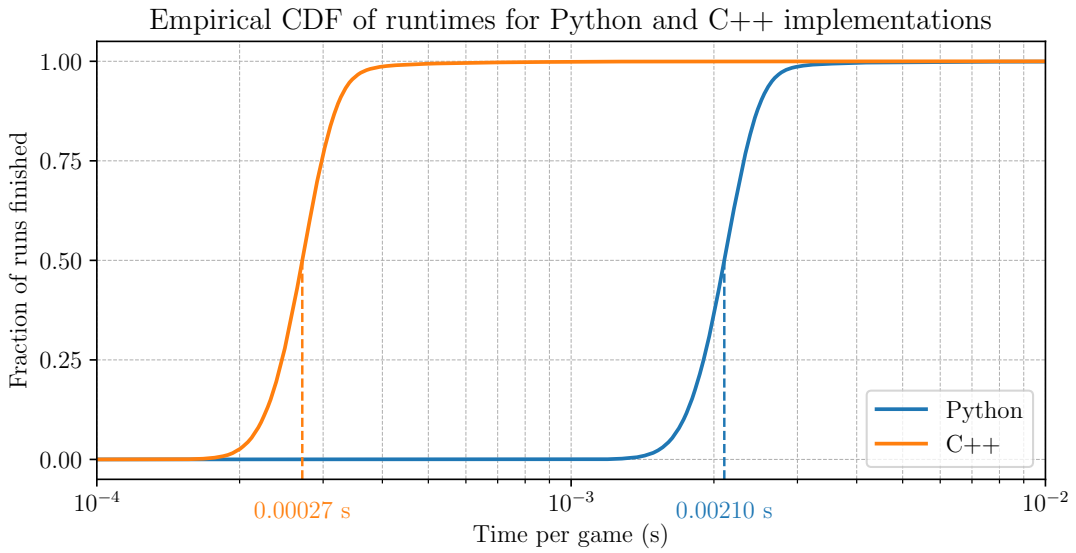


Figure 5.2: Empirical CDFs of the time used to solve boards with random play, split on C++ and Python implementations, based on 100 000 randomly generated boards. Median time to solve a board was 0.00027 seconds with C++, and 0.00210 seconds with Python.

All code used in this project is available in the GitHub repository linked in Appendix B.1. This also includes a graphical user interface (GUI) that anyone can use to play the game and obtain recommendations from one of our solvers at each turn. Examples of this GUI are shown in Appendix B.2, and a detailed explanation of how to use it is in the README file in the GitHub repository.

An appropriate way of evaluating the performance of our models and solvers is to compare their solutions to the best scores obtained by anyone in Norway on daily NRK boards. NRK does not keep previous boards available on their website, and therefore we have stored 100 daily boards over the period from January 27$^{\text{th}}$ to May 20$^{\text{th}}$ 2025, to use for testing purposes. For reproducibility, these boards are available in the GitHub repository.

5.2 Self-made Heuristics

Heuristics are game-specific strategies that perform well according to the purpose of the game. In the case of *Former*, heuristics are strategies that use fewer moves to clear the board than guessing randomly. In this section, we explain our choice of self-made heuristics, and how approximate value and policy functions are defined based on them.

We use two types of self-made heuristics. The first is the greedy strategy of always removing the largest group from the board. The second is a greedy look-ahead strategy that minimizes the number of possible actions n steps ahead.

For the largest-group-first strategy, we define a corresponding value function that is the number of remaining shapes on the board, which we denote $M(s)$,

$$v_{\text{largest}}(s) = M(s). \tag{5.1}$$

This is not a particularly good approximation to the optimal value function, but it may still perform well as guidance for search techniques if the actual minimum number of moves remaining is sufficiently correlated with the number of shapes left on the board.

For the n look-ahead heuristic, we define the value function to be the lowest attainable number of groups n steps ahead. Hence, in the base case ($n = 0$), the corresponding value function is the current number of groups,

$$v_0(s) = G(s). \tag{5.2}$$

For $n > 0$, we minimize this value over all possible sequences of actions $\{a_1, \ldots, a_n\}$. Recall that $s' = \tau(s, a)$ denotes the transition function from state s to s' through action a. Then, we can define the n look-ahead value function of some state s as

$$v_n(s) = \min_{a_1,\ldots,a_n} G\Big(\underbrace{\tau\big(\cdots \tau(s,\, a_1) \cdots ,\, a_n\big)}_{k \text{ actions ahead}}\Big). \tag{5.3}$$

Again, although this value function generally does not give the actual remaining number of moves, it may still work well as guidance for search techniques if the number of groups n steps ahead is sufficiently correlated with the actual optimal value function. Due to the exponential increase in the number of possible action combinations, it is only worth checking the first few steps ahead. Thus, we implement the n look-ahead heuristic for $n = 1$, $n = 2$ and $n = 3$.

Based on the value functions, we craft deterministic policy functions for each heuristic. That is,

$$\pi(a \mid s) = \begin{cases} 1 & \text{if} \quad v(\tau(s,a)) = \min_{a'} v(\tau(s,a')), \\ 0 & \text{otherwise.} \end{cases}$$

Although these do not allow any exploration on their own, using them in MCTS automatically provides some exploration, and thus we allow these policies to be deterministic.

We evaluate the self-made heuristics through *greedy play* with their respective policy functions. That is, at each step t and state s_t, we choose the action a_t that maximizes the policy function,

$$a_t = \arg\max_a \pi(a \mid s_t),$$

and record the number of moves used and the mean runtime to clear boards. Although the models are not intended to solve boards on their own through greedy play, using few

moves and having a low standard deviation indicates that a model is accurate and can serve as a reliable guide when incorporated into search techniques.

The evaluation of each heuristic is done using two separate datasets. The first contains 1 000 boards generated from the discrete uniform distribution with fixed seed `np.random.seed(22)`. We use a large number of boards to obtain statistically reliable measures of the performance of each model, which we compare to determine which heuristics are feasible to use with search techniques. The second dataset is the 100 daily boards published by NRK, which we use to monitor how close our heuristics come to the best-known solutions.

5.3 Supervised Learning

In this section, we cover the exact methods used to train policy and value networks with supervised learning. Recall that, similar to the self-made heuristics, these networks are models that we later combine with MCTS and beam search. Hence, the purpose of the networks is not to solve *Former* boards on their own, but rather to suggest promising actions and evaluate board difficulty. First, we explain the process of generating data, before we provide network architectures and explain the hyperparameter tuning process. Then, we give details of the training procedure, which includes our choice of training and validation loss, before explaining how networks are evaluated.

5.3.1 Data Generation

We do not have many official daily boards at our disposal, and the ones we have are more interesting to use for testing purposes. Therefore, we generate our own data to train neural networks. This is done in 3 steps:

1. Generate a board based on the discrete uniform distribution.

2. Find an "expert" solution to the board using beam search with a fixed beam width and a reasonable heuristic as guidance.

3. Save each state-action pair along with the number of moves it took to solve the board from that state. For example, if the board is cleared in 15 moves, we obtain 15 tuples of the form
$$(\mathbf{x}_i, a_i, T_i),$$
where $\mathbf{x}_i = \psi(s_i)$ is the tensor representation of a board, a_i is the expert move and T_i is the number of moves remaining. In our implementation, the tensor $\mathbf{x}_i$ represents a one-hot encoded board,
$$\mathbf{x}_i \in \{0,1\}^{9\times 7\times 5}.$$
In one-hot encoding, a 1 at index (n, m, r) means that there is a shape of type r at grid point (n, m). Here, $r \in \{0, 1, 2, 3, 4\}$, where $r = 4$ represents an empty grid point.

The motivation for using beam search to generate data is that it acts as an expert playing the game, thus it provides high-quality data that the networks can be trained on. Although it is not guaranteed to find the optimal solution for any board, using a solver gives better data than greedy play with the heuristic model on its own. Hence, by

mimicking the behavior of the solver, the networks may turn out better than the heuristics. This approach of training on expert data is typically called *imitation learning* (Ho & Ermon, 2016), and has proven useful for training neural networks to make predictions on games such as *Go*, where perfect data is not available (Silver et al., 2016).

We use a beam search solver to generate data instead of MCTS since it gave the best results in preliminary studies with self-made heuristics as guidance. To assess which heuristic and corresponding beam width to use, we run a quick analysis of the performance of beam search with the 1 and 2 look-ahead heuristics, which we found to be the most promising (more details on this in Section 6.1). Since we need a substantial amount of data, we set a time limit of

$$t_{\max} \approx 1 \text{ second}$$

to generate one play-through. The analysis is based on 1 000 randomly generated *Former* boards, for which we record the runtime and number of moves used by beam search with both heuristics and a w_{beam} that gives runtime close to 1 second. The results are presented in Table 5.1. From this we observe that, given a time constraint of 1 second, the 1 look-ahead heuristic with $w_{\text{beam}} = 128$ on average finds better solutions. Consequently, this is the configuration that we use to generate expert training data.

Table 5.1: Average solution length and runtime for beam search with 1 look-ahead heuristic with $w_{\text{beam}} = 128$, and 2 look-ahead heuristic with $w_{\text{beam}} = 16$.

Heuristic	**Beam width**	**Mean moves**	**Mean runtime (s)**
1 look-ahead	128	14.8	0.86
2 look-ahead	16	15.1	1.08

In total, we generated 9.083.686 data samples using beam search with the 1 look-ahead heuristic and $w_{\text{beam}} = 128$. This amount of data is assumed to be sufficient for the networks to learn general strategies, without making training times impractically long. The 9 million samples correspond to approximately 600.000 games, from which we remove the games that used more than 20 moves, as these are assumed not to represent good gameplay. This only constituted 364 samples. After splitting the games into 90% training and 10% validation, we randomize the order of the data samples to avoid correlation between samples from the same game affecting the results.

In Figure 5.3, we provide an overview of the number of samples in the dataset per number of remaining moves. Here, we observe that the data are biased towards lower move counts, which is because of the generation process: all games provide a sample with, say, 5 moves remaining, but only a 20-move game provides a sample with 20 moves remaining. Biased data may cause the networks to obtain lower accuracy on more difficult boards, as they are rarely encountered during training. An option to handle this issue is to upsample the data, meaning that we duplicate observations with 14 moves, 15 moves, and so on, to obtain an even distribution. However, this approach may cause other issues. Data generated by beam search are inherently biased towards a higher number of remaining moves than the optimal solutions, since the solver is not perfect. Thus, there are two biases acting against each other: bias towards lower move counts as there are more samples where fewer moves remain, and bias towards higher move counts since beam search makes mistakes. We cannot know how to perfectly balance these biases without knowing what the optimal solutions are, and therefore we leave the data as is.

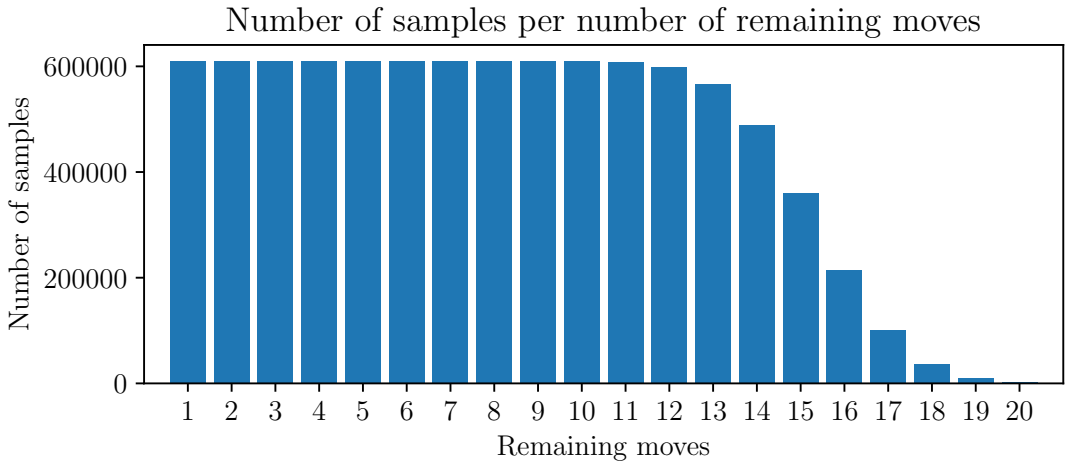


Figure 5.3: The number of samples in the dataset per number of moves remaining.

5.3.2 Neural Network Architectures

Now that we have covered data generation, we switch focus to the networks themselves. In this subsection, we introduce the network architectures that we use. In total, we make 8 models with the supervised learning approach: four value networks and four policy networks. The purpose of training four of each is to test how the number of layers d and the number of filters per layer w affect the overall performance of both types of networks. We do not use Bayesian optimization to tune w and d since other network hyperparameters are highly correlated with the network architecture, which may cause slow convergence of the optimization algorithm (Snoek et al., 2012). Moreover, for a fixed dataset, we would expect the network with the highest w and d to perform better, which does not take the runtime of network evaluations into account. Therefore, we fix

$$w \in \{32, 64\} \quad \text{and} \quad d \in \{5, 10\},$$

which should be sufficient to show the impact of w and d, while keeping the networks light enough for fast evaluation. We denote policy and value networks with d layers and w filters by

$$f_{\pi,(w,d)} \quad \text{and} \quad f_{v,(w,d)},$$

respectively.

In Table 5.2 we show the network architectures and the parameters relevant to each layer. Both policy and value networks have the same *backbone*, consisting of d convolutional blocks. In the first block, we use a kernel size $k = 7$ to enable the networks to recognize larger spatial patterns from the input grid, as discussed in Section 3.1. In the next $d-1$ convolutional blocks, we set $k = 3$, which is a common choice to enable further feature extraction while keeping the network lightweight for fast evaluations (Silver et al., 2016). After the backbone of d convolutional blocks, the policy and value network architectures differ. Value networks employ a GAP layer, followed by a fully connected layer with one neuron and the linear activation function (3.5). Policy networks use a convolutional layer with kernel size $k = 1$ to preserve spatial features, followed by the softmax activation function (3.4) to output probabilities over each of the 9×7 grid spaces. In our formulation of *Former* (Section 2.3.1), a board consists of $G(s)$ possible actions, one for each of the $G(s)$ groups on the board. Hence, the output of policy networks should assign

one probability to each group, and not one probability to each grid point. To obtain this, we perform *masking*, which consists of two steps:

1. Setting the probability of all board spaces with no shapes to 0, and standardizing.
2. For each of the $G(s)$ groups, sum up the probabilities over all shapes in that group, and assign that probability to only the first action, as a representative of the group.

Masking is important because, when the beam search solver generated data, each group was represented by the first grid point in that group. So, to obtain the correct measure of network accuracy, the network should output a distribution over the same actions, where a prediction is considered correct as long as the network assigned the highest cumulative probability to the correct group. If we do not mask, the network must first learn to predict the correct action in a group, which may hinder it from learning actual game strategies.

Table 5.2: Layers in policy and value networks, and the kernel sizes of each convolutional layer.

Policy network architecture			
Layer(s)	**Input size**	**Output size**	**Kernel size**
1 convolutional block	$9 \times 7 \times 5$	$9 \times 7 \times w$	$k = 7$
$d - 1$ convolutional blocks	$9 \times 7 \times w$	$9 \times 7 \times w$	$k = 3$
1 convolutional block	$9 \times 7 \times w$	9×7	$k = 1$
Softmax activation	9×7	9×7	—
Masking	9×7	$G(s)$	—
Value network architecture			
Layer(s)	**Input size**	**Output size**	**Kernel size**
1 convolutional block	$9 \times 7 \times 5$	$9 \times 7 \times w$	$k = 7$
$d - 1$ convolutional blocks	$9 \times 7 \times w$	$9 \times 7 \times w$	$k = 3$
1 GAP layer	$9 \times 7 \times w$	w	—
1 fully connected layer	w	1	—
Linear activation	1	1	—

5.3.3 Hyperparameter Tuning

For each of the eight networks, we tune the hyperparameters

$$\boldsymbol{\xi} = (n_{\text{batch}}, \eta, \beta),$$

that is, batch size, learning rate in the Adam optimizer, and weight decay in the loss function. We use the TPE surrogate model with the EI acquisition function, as discussed in Section 3.2.4. The assumed domains of the hyperparameters are presented in Table 5.3.

We use a subset of 2.000.000 data samples from the original data to perform hyperparameter tuning. This subset has a split of 90% training and 10% validation. A network is trained for 5 epochs before the validation loss is calculated, and Bayesian optimization is used to determine the next choice of parameters, as described in Section 3.2.4. In total,

Table 5.3: Hyperparameter domains.

Hyperparameter	Symbol	Domain
Batch size	n_{batch}	$\{32, 64, 128, 256\}$
Learning rate	η	$[10^{-6}, 10^{-2}]$
Weight decay	β	$[10^{-6}, 10^{-2}]$

we run each network for 50 such iterations, and choose the set of hyperparameters that results in the lowest validation loss.

For value networks, we use the MSE validation loss during hyperparameter tuning,

$$\mathcal{L}_{\text{val},v}(\boldsymbol{\xi}) = \mathcal{L}_{\text{MSE}}(\boldsymbol{\theta}(\boldsymbol{\xi}); B_{\text{val}}).$$

For policy networks, we use Top-1 accuracy as defined in (3.12), since this gives an intuitive measure of network performance. To be precise, we use the negative Top-1 metric since hyperparameter tuning is formulated as a minimization problem, that is,

$$\mathcal{L}_{\text{val},\pi}(\boldsymbol{\xi}) = -\text{Top}_1\left(\boldsymbol{\theta}(\boldsymbol{\xi}); B_{\text{val}}\right).$$

Hyperparameter importance is calculated using (3.15), with the procedure explained in Section 3.2.4.

5.3.4 Training and Validation

After hyperparameter tuning, we train each of the four policy and four value networks on the full training set with the respective hyperparameters. We use MSE loss and cross-entropy loss for training value and policy networks, respectively. As explained in Section 3.2.2, a regularization term is also added to avoid overfitting. Thus, the loss functions used for training neural networks are given by (3.10) for policy networks and (3.11) for value networks. We train for a total of 15 epochs, after which we choose the networks associated with the epochs of lowest validation losses.

Cross-entropy and MSE are also used for validation, as discussed in Section 3.2.3. We do not add weight decay, since we want a measure of the exact performance of each network, regardless of parameter size. Validation loss is calculated using the same batch size as in training, which is for the sake of computational efficiency. Validation batches are kept identical at each epoch and for each network trained, to ensure that the metrics are comparable.

5.3.5 Evaluation

We evaluate the performance of the networks through greedy play on the same two test sets used to evaluate self-made heuristics, that is, one with 1 000 randomly generated boards and one with 100 official boards. Let $\left[f_\pi(s;\boldsymbol{\theta})\right]_a$ denote the probability assigned to action a in state s by the policy network. Then, greedy play with policy networks involves choosing the action assigned the highest probability at each step,

$$a_t = \arg\max_a \left[f_\pi(s_t;\boldsymbol{\theta})\right]_a.$$

With value networks, we look one step ahead and choose the action leading to the state with the lowest estimated remaining number of moves,

$$a_t = \arg\min_a f_v(\tau(s_t, a); \boldsymbol{\theta}),$$

where τ is the transition function defined in Section 2.3.1.

The purpose of evaluating networks through greedy play is to compare how well they perform on *Former* boards. We do not expect them to solve many boards perfectly, as their purpose first and foremost is to guide the solvers, and not to find the best-known solutions by themselves. We return to how networks are incorporated into search techniques to make solvers in Section 5.5. Before this, we explain the implementation details of the final approach used to create models: PPO.

5.4 Proximal Policy Optimization

In this section, we cover the methodology that we use to train neural networks with PPO. Recall from Section 3.3 that PPO uses an actor-critic network, which consists of a common backbone, a policy head, and a value head. Our intention is to train such networks with PPO, extract the policy and value heads with the common backbone as separate actor and critic networks, $\pi_{\boldsymbol{\theta}}$ and $v_{\boldsymbol{\phi}}$, and finally use them in solvers to search for solutions to *Former* boards. We come back to how they are incorporated into search techniques in the next section. Here, we first give our choice of actor-critic network architectures, then the hyperparameters used, and finally details of the training procedure of PPO models.

For ease of implementation, we use existing Python libraries that support PPO: the Gymnasium environment interface (Towers et al., 2024) and the Stable-Baselines3 algorithm library (Raffin et al., 2021). To use these, we first implement *Former* as a custom Gymnasium environment. All relevant code is included in the GitHub repository of Appendix B.1.

5.4.1 Actor-critic Network Architecture

In this subsection, we cover the network architectures used in PPO models. These are similar to those used in the supervised learning setting (Section 5.3.2), but due to the different structure of dual-head networks and for ease of implementation, we only set the backbone of the network to our liking, and use the actor and critic heads that are standard in the Stable-Baselines3 implementation of PPO (Raffin et al., 2021). The layers in the common backbone, in the actor head and in the critic head are shown in Table 5.4.

We mask the output of the actor network, which we found to speed up the convergence of the training process. Masking is done slightly differently from the supervised case explained in Section 5.3.2. There is no reason for the actor to learn that only the first shape of a group is a valid action, and as a result, masking too much of the board will lead to slow or no convergence. What we found worked the best was to mask only the actual illegal moves, that is, where there are no shapes, and allow the PPO model to choose among all grid points with shapes in them. In Table 5.4, $M(s)$ denotes the number of shapes left in board s.

Table 5.4: Layers in the actor-critic networks, with d convolutional blocks and w filters per layer. The network backbone is shared, and splits into the actor head, which outputs a probability distribution over the $M(s)$ shapes on the board, and the critic head, which outputs a scalar value.

Network backbone			
Layer(s)	**Input size**	**Output size**	**Kernel size**
1 convolutional block	$9 \times 7 \times 5$	$9 \times 7 \times w$	$k = 7$
$d - 1$ convolutional blocks	$9 \times 7 \times w$	$9 \times 7 \times w$	$k = 3$
Flatten	$9 \times 7 \times w$	$63 \cdot w$	—
1 fully connected	$63 \cdot w$	256	—
ReLU	256	256	—
Actor (policy) head			
Layer(s)	**Input size**	**Output size**	**Kernel size**
1 fully connected	256	63	—
Softmax	63	63	—
Masking	63	$M(s)$	—
Critic (value) head			
Layer(s)	**Input size**	**Output size**	**Kernel size**
1 fully connected	256	1	—

5.4.2 Hyperparameters and Reward Shaping

Here, we give the hyperparameters used and define the reward signals obtained by the PPO agent during training. PPO models are trained for a period of several days to achieve convergence, and therefore we do not extensively explore all hyperparameters. Instead, we rely on the literature and game knowledge to fix a set of hyperparameters, and as long as they cause convergence, we are satisfied with the choice.

All hyperparameters introduced in Section 3.3, as well as the reward per action, are included in Table 5.5. The ones relevant for training that we did not cover are set to the standard values in the PPO implementation of StableBaselines-3 (Raffin et al., 2021). For each of the hyperparameters given, we also provide a brief explanation.

The initial learning rate η_0 is set to the Stable-Baselines3 standard of $\eta_0 = 3.0 \times 10^{-4}$. We use a linearly annealing learning rate scheduler, meaning that the learning rate $\eta(t)$ starts at $\eta(0) = \eta_0$ and steadily decrease as time goes by,

$$\eta(t) = \eta_0 \left(1 - \frac{t}{N_{\max}}\right).$$

Using a linearly annealing learning rate has shown to provide stable convergence and improved performance of the networks obtained from PPO (Mnih et al., 2016; Schulman et al., 2017).

As discussed in Section 2.3.1, we set the discounting factor $\gamma = 1$ and the reward per action $R_t = -1$, since each move in a solution is worth the same to the final solution. We do, however, set $\lambda = 0.95$ to reduce the large variance in the rewards obtained, as

Table 5.5: Hyperparameters and reward per action in PPO models.

Hyperparameter	**Symbol**	**Value / Notes**
Initial learning rate	η_0	3.0×10^{-4}
Learning rate scheduler	–	Linear (η_0 to 0)
Discount factor	γ	1
Reward per action	R_t	-1
TD parameter	λ	0.95
PPO clip factor	ε	0.20
Samples collected per environment	n_{steps}	1048
Number of parallel environments	n_{envs}	16
Batch size	n_{batch}	256
Number of PPO epochs	n_{epoch}	10
Value-loss coefficient	c_{val}	0.5
Entropy coefficient	c_{ent}	0.0
Maximum steps	$N_{\max}$	2×10^9

discussed in Section 3.3.2. The value $\lambda = 0.95$ is the standard choice in the Stable-Baselines3 implementation. We set the clip factor $\varepsilon = 0.2$, which is a PPO standard used by Schulman et al. (2017).

The hyperparameters n_{steps}, n_{envs}, n_{batch} and n_{epoch} are related to the data used in each network update. Here, we use $n_{\text{steps}} = 1048$ and $n_{\text{envs}} = 16$, meaning that we collect data in parallel over 16 environments, each collecting 1048 data samples, leading to a total of $N = 16 \cdot 1.024 = 16.384$ data samples. This data is then split into batches of size $n_{\text{batch}} = 256$, and trained for $n_{\text{epoch}} = 10$ epochs. The batch size is set to a larger value than we used in the supervised learning setting to avoid overfitting on the relatively small dataset, and since we plan on running the training procedure for a longer period of time.

The value loss and entropy bonus coefficients, $c_{\text{val}} = 0.5$ and $c_{\text{ent}} = 0$, are also set to the standard values of Stable-Baselines3. This means that we do not give any entropy bonus to the model. This is partially because tuning the entropy bonus properly requires running the algorithm several times, and partially because we plan on implementing the algorithms in search techniques, anyway, which inherently provide exploration even if the actor is deterministic.

The maximum number of steps $N_{\max} = 2 \times 10^9$ is set so that the algorithms can train for several days before stopping. One step is equivalent to one action. In practice, this number is large enough so that the training is stopped by some time limit before reaching the step limit, which we discuss in the next subsection.

5.4.3 Training

We train PPO models for a fixed time limit. This time limit is mainly determined by the computational resources we have at hand. For each PPO-based network we train, we use 1 GPU for network updates and 16 CPUs for data collection, split over the $n_{\text{envs}} = 16$ environments. To obtain such computational power, we use the Idun supercomputer provided by the Norwegian University of Science and Technology. Training is carried out with a time limit of 4 days.

Since training each model requires a significant amount of computational power over

a longer period of time, we only train two networks. To make these comparable to the supervised learning models, we set

$$(w, d) \in \{(32, 5), (64, 10)\},$$

which we refer to as the *small* and *large* networks, respectively. The actor and critic models of each network are denoted

$$\pi_{\boldsymbol{\theta},(w,d)} \quad \text{and} \quad v_{\boldsymbol{\phi},(w,d)}.$$

5.4.4 Evaluation

After training, we extract the dual-head networks, and treat the actor and critic of each network as separate models. We evaluate them using the exact same greedy play approach as with supervised learning models, explained in Section 5.3.5.

Like self-made heuristic and supervised learning models, the purpose of training PPO models is not to have them find the best solutions to all boards. They predict what actions appear promising and how many moves remain for a given board, which is incorporated into search techniques to perform the actual solving. This is the topic of the next and final section of this chapter.

5.5 Search Techniques

Now that we have covered each of the three approaches we use to create models, we explain how they are combined with search techniques to make solvers. Recall from Sections 5.2, 5.3 and 5.4 that each approach provides policy and value models, which suggest reasonable actions and the remaining number of moves from some board. Also, recall from Chapter 4 that MCTS relies on a policy function to suggest promising actions from a single board, and that beam search relies on a value function to compare the predicted remaining moves across many boards. Thus, we split between two types of solvers: MCTS solvers, which combine MCTS with policy models, and beam search solvers, which combine beam search with value models. In this section, we explain the implementation details of the two search techniques and list the models combined with each of them: first for MCTS and then for beam search. Afterwards, we describe how the solvers are evaluated on two datasets.

5.5.1 MCTS Implementation Details

The fixed parameters used in our implementation of MCTS are included in Table 5.6. The exploration parameter c_{puct} is set to 10, which we found to give sufficient exploration to discover good solutions. This parameter does not require much tuning since the minimum exploration limit, $N_{\min}$, already enforces some exploration. We set $N_{\min} = 10$, which is similar to what was done by Silver et al. (2016) and Schadd et al. (2008).

A single search with MCTS lasts for a predetermined number of seconds $t_{\max}$. The choice of $t_{\max}$ depends on the size of the test set we evaluate on, which we come back to in Section 5.5.3.

In total, we use 8 different policy models in MCTS. These are listed in Table 5.7.

Table 5.6: The parameters used in MCTS.

Parameter	Symbol	Value
Exploration constant	c_{puct}	10
Minimum visits from root node	$N_{\min}$	10

Table 5.7: Policy models used in MCTS solvers.

Model	Explanation	Covered in
π_1	Policy function for the 1 look-ahead heuristic	Section 5.2
π_2	Policy function for the 2 look-ahead heuristic	
$f_{\pi,(32,5)}$	Policy network with $w = 32$, $d = 5$	Section 5.3
$f_{\pi,(32,10)}$	Policy network with $w = 32$, $d = 10$	
$f_{\pi,(64,5)}$	Policy network with $w = 64$, $d = 5$	
$f_{\pi,(64,10)}$	Policy network with $w = 64$, $d = 10$	
$\pi_{\boldsymbol{\theta},(32,5)}$	PPO actor with $w = 32$, $d = 5$	Section 5.4
$\pi_{\boldsymbol{\theta},(64,10)}$	PPO actor with $w = 64$, $d = 10$	

5.5.2 Beam Search Implementation Details

Beam search is implemented as an anytime algorithm, inspired by Odland (2024). This means that we let the algorithm run for increasing values of w_{beam}, and stop once an iteration exceeds the time limit $t_{\max}$. In particular, we initially search with a beam width of 1, and increase by a factor of 2 each time,

$$w_{\text{beam}} = 1, 2, 4, 8, 16, \ldots,$$

until the last search exceeds the time limit.

We incorporate 8 value models into beam search, which are listed in Table 5.8.

Table 5.8: Value functions used in beam search solvers.

Model	Explanation	Covered in
v_1	Value function for the 1 look-ahead heuristic	Section 5.2
v_2	Value function for the 2 look-ahead heuristic	
$f_{v,(32,5)}$	Value network with $w = 32$, $d = 5$	Section 5.3
$f_{v,(32,10)}$	Value network with $w = 32$, $d = 10$	
$f_{v,(64,5)}$	Value network with $w = 64$, $d = 5$	
$f_{v,(64,10)}$	Value network with $w = 64$, $d = 10$	
$v_{\boldsymbol{\phi},(32,5)}$	PPO critic with $w = 32$, $d = 5$	Section 5.4
$v_{\boldsymbol{\phi},(64,10)}$	PPO critic with $w = 64$, $d = 10$	

5.5.3 Evaluation

We evaluate the solvers on the same datasets as the models: one with 1 000 randomly generated boards, and one with 100 official boards published by NRK. The solvers are

evaluated through *search*, where each solver is allowed to search for solutions for a predetermined time limit, exploring the state space based on the particular model and search technique it consists of.

As with the models, the purpose of evaluating the solvers on the dataset containing 1 000 boards is to obtain statistically reliable measures of how well the solvers perform and compare their performance to that of greedy play with the models. Since the size of this test set is relatively large, we set a time limit $t_{\max} = 10$ seconds per board per solver. During these 10 seconds, we monitor all the solutions found by each solver and the respective time stamps they were found. This allows us to calculate the best-so-far solution curves, which for each solver show how the mean number of moves used to clear the boards decreases as the search time increases.

Evaluating on the set of 100 daily boards allows us to compare the performance of our solvers to the best-known solutions to each board. For this purpose, we set a time limit $t_{\max} = 60$ seconds, and use all solvers on all boards. Thereafter, we choose one solver to use for two additional purposes: First, we monitor the number of unique and equivalent best solutions that exist for a couple handpicked boards, and second, we set a time limit of 24 hours, and check whether there exist better solutions than what is known to some of the daily boards.

CHAPTER

SIX

RESULTS AND DISCUSSION

This chapter presents the results from using self-made heuristics and neural networks with search techniques to solve *Former* boards. The results follow the same order as the methodology presented in Chapter 5. In Sections 6.1, 6.2 and 6.3, we present the results from the three approaches to approximating π^* and v^*: self-made heuristics, supervised learning, and PPO, respectively. After these, in Section 6.4, we show the results of testing each solver on randomly generated and official *Former* boards.

6.1 Self-made Heuristics

To evaluate our self-made heuristic models, we generated a test set of 1 000 boards by sampling each shape from a discrete uniform distribution. Each board was solved by a random baseline strategy, that is, by selecting moves randomly, and by greedy play with each self-made heuristic in turn. For each heuristic, we record the number of moves to solve each board, and the corresponding runtimes.

Table 6.1: Aggregate performance of random gameplay and greedy play with self-made heuristics on 1 000 randomly generated boards.

Heuristic	**Mean moves**	**Std. Dev.**	**Time per board (ms)**
Random	28.67	3.72	0.27
Largest group	24.99	4.12	0.61
1 look-ahead	18.54	2.64	5.7
2 look-ahead	16.21	1.91	110
3 look-ahead	15.48	1.80	2840

Table 6.1 summarizes, for each heuristic, the mean moves to clear a board, the standard deviation in the number of moves, and the average runtime per board. Figure 6.1 shows the full distribution of moves per board for each heuristic. Unsurprisingly, we observe that the larger the look-ahead horizon n, the lower the mean number of moves for the n look-ahead heuristics. The runtime of these, however, increases exponentially, reaching almost 3 seconds per board for the 3 look-ahead heuristic. Despite obtaining the lowest mean number of moves, having a runtime of 3 seconds is infeasible to use in a search technique. What is perhaps more surprising, is the high move count for the largest-group-first heuristic. Although it seems like a feasible strategy, it does not perform significantly

better than guessing randomly, and in Figure 6.1 we observe that the worst-case solution in fact is worse than random play. This is consistent with what we showed in Figure 2.3 of Chapter 2, and is likely caused by there being many small groups remaining towards the end of the play-through.

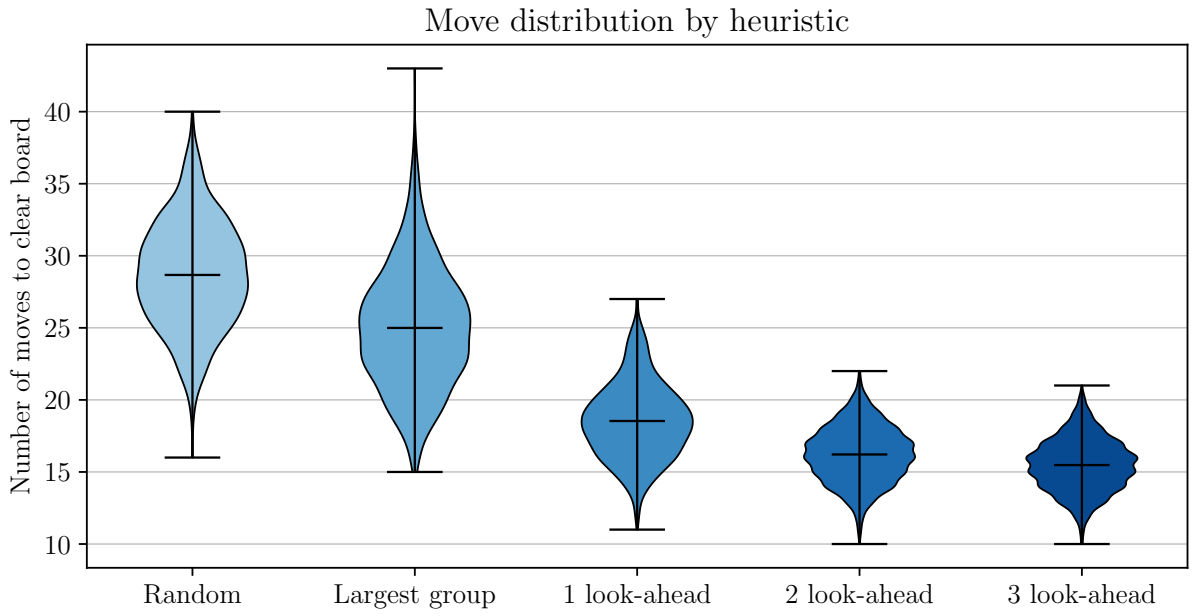


Figure 6.1: Distribution of move count for random and greedy gameplay using self-made heuristics, over 1 000 randomly generated *Former* boards.

To further assess the accuracy of the self-made heuristics, we use them in greedy play on the set of 100 daily boards that NRK published between January 27$^{\text{th}}$ and May 20$^{\text{th}}$, and compare how much the solutions found deviate from the best-known solutions. Let Δ denote this deviation, which is the number of extra moves each heuristic uses compared to the best-known solutions. The results are presented in Table 6.2. The 3 look-ahead heuristic performs the best, followed by the 2 look-ahead one, which found the best solutions to 10 and 2 out of the 100 boards, respectively. Although the purpose of the heuristics is not to find the best-known solutions, performing well on the daily boards is a good indication of how they will perform when combined with search techniques. In addition to using few moves, however, a good model also has a low runtime. Thus, based on the overall performance of the self-made heuristic models on both datasets, we only use the 1 and 2 look-ahead heuristics in solvers, as these give reasonable trade-offs between a low mean move count and a low runtime.

6.2 Supervised Learning

In this section, we provide the results obtained with the neural networks trained using supervised learning on self-generated data. We first show the outcomes of hyperparameter tuning, where we have found appropriate combinations of parameters for training four policy and four value networks. Thereafter, we show the results from the training procedures of these networks, before we evaluate the networks through greedy play on the same two datasets that we used to evaluate the self-made heuristics.

Table 6.2: The number of boards per solution found by each heuristic model through greedy play, measured in difference to the best-known solution, Δ.

Number of solutions per Δ (heuristic models)						
Model	$\Delta = 0$	$\Delta = 1$	$\Delta = 2$	$\Delta = 3$	$\Delta = 4$	$\Delta \geq 5$
Random	0	0	0	0	0	100
Largest group	0	0	1	0	0	99
1 look-ahead	0	4	9	13	23	51
2 look-ahead	2	23	25	31	13	6
3 look-ahead	10	26	37	15	7	5

6.2.1 Hyperparameter Tuning

This subsection presents the results from hyperparameter tuning of eight neural networks based on Bayesian optimization. We first show the best value and policy networks for each combination of $d \in \{5, 10\}$ convolutional blocks and $w \in \{32, 64\}$ filters per layer, before we present hyperparameter importance and convergence. The parameters and the respective domains we tune over are given in Table 5.3.

Value networks are compared based on MSE, and policy networks based on Top-1 accuracy. Table 6.3 presents the best-performing models. We observe that the policy network typically perform better with higher learning rates and higher batch size compared to value networks, and that weight decay is set to a low value for all networks.

Table 6.3: Parameters in the best policy and value networks according to Top-1 accuracy and validation MSE, for each combination of $w \in \{32, 64\}$ and $d \in \{5, 10\}$.

Best policy networks				
Network	Learning rate	Batch size	Weight decay	Top-1
$f_{\pi,(32,5)}$	0.0028	128	$2.5 \cdot 10^{-6}$	0.521
$f_{\pi,(32,10)}$	0.0049	256	$1.8 \cdot 10^{-6}$	0.582
$f_{\pi,(64,5)}$	0.0045	128	$1.6 \cdot 10^{-6}$	0.555
$f_{\pi,(64,10)}$	0.0031	128	$1.0 \cdot 10^{-6}$	0.621
Best value networks				
Network	Learning rate	Batch size	Weight decay	MSE
$f_{v,(32,5)}$	0.00073	64	$5.9 \cdot 10^{-6}$	0.488
$f_{v,(32,10)}$	0.0022	128	$2.0 \cdot 10^{-6}$	0.484
$f_{v,(64,5)}$	0.00045	64	$6.6 \cdot 10^{-6}$	0.441
$f_{v,(64,10)}$	0.00049	32	$1.5 \cdot 10^{-6}$	0.430

Figures 6.2 and 6.3 show the convergence of the optimization algorithm and the permutation importance of hyperparameters, respectively, for the policy and value networks with $w = 64$ and $d = 5$. These are calculated based on the procedure explained in Section 3.2.4. The other networks gave very similar results, so for the sake of keeping this section concise we only display results for the networks with $w = 64$ and $d = 5$. The best-so-far lines in Figure 6.2 show the lowest loss found up until each trial, which seem

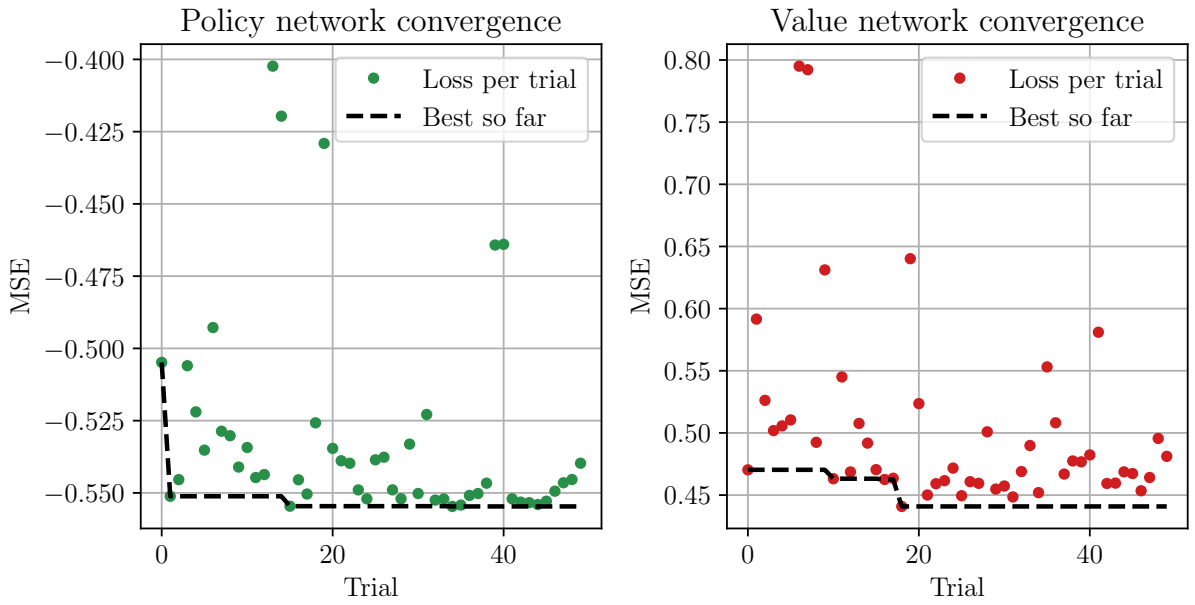


Figure 6.2: Convergence of the hyperparameter tuning algorithm for a policy network (left display) and a value network (right display), both with $w = 64$ and $d = 5$.

to have converged sufficiently. From the hyperparameter importance in Figure 6.3 we observe that the learning rate is the most important hyperparameter to tune appropriately.

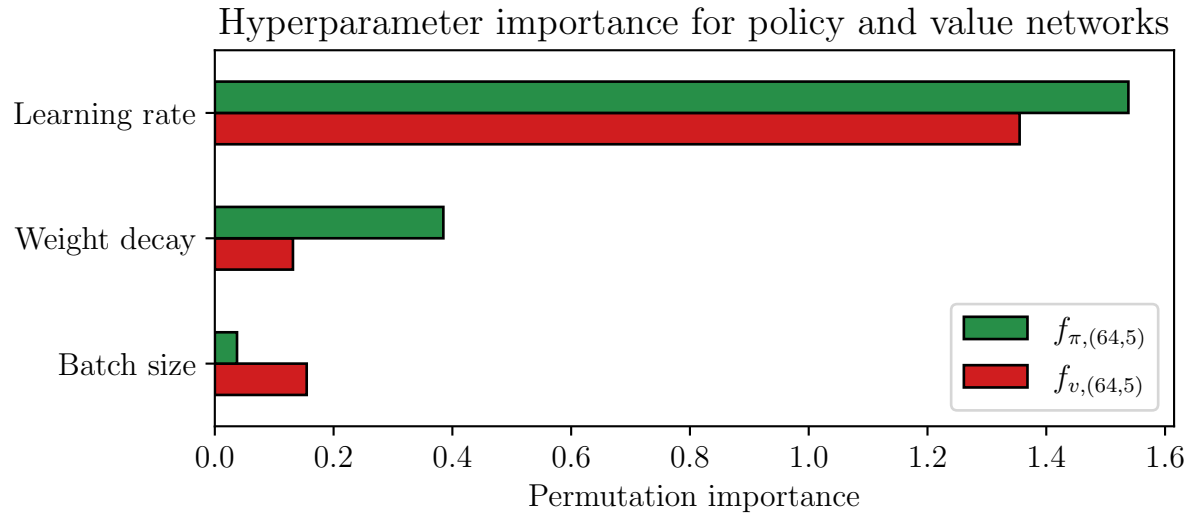


Figure 6.3: Permutation importance from hyperparameter tuning for the policy and value networks with $d = 5$ convolutional layers and $w = 64$ filters per layer.

6.2.2 Training and Validation

We now show the learning dynamics of the eight networks, starting with the four policy networks. Figure 6.4 displays the training and validation losses over epochs for the four policy networks. An important observation here is that the validation losses are slightly lower than the training losses, especially in the first epochs. We assume this to be due to

the regularization term $\mathcal{R}(\boldsymbol{\theta})$, which is added during training but not in validation. As the number of epochs increases, the training loss stabilizes, indicating that the training procedure has converged. The validation losses do not increase towards the end, hence there are no signs of overfitting. For further analysis, we use the policy networks corresponding to Epoch 15 in Figure 6.4, as these provide the lowest validation loss.

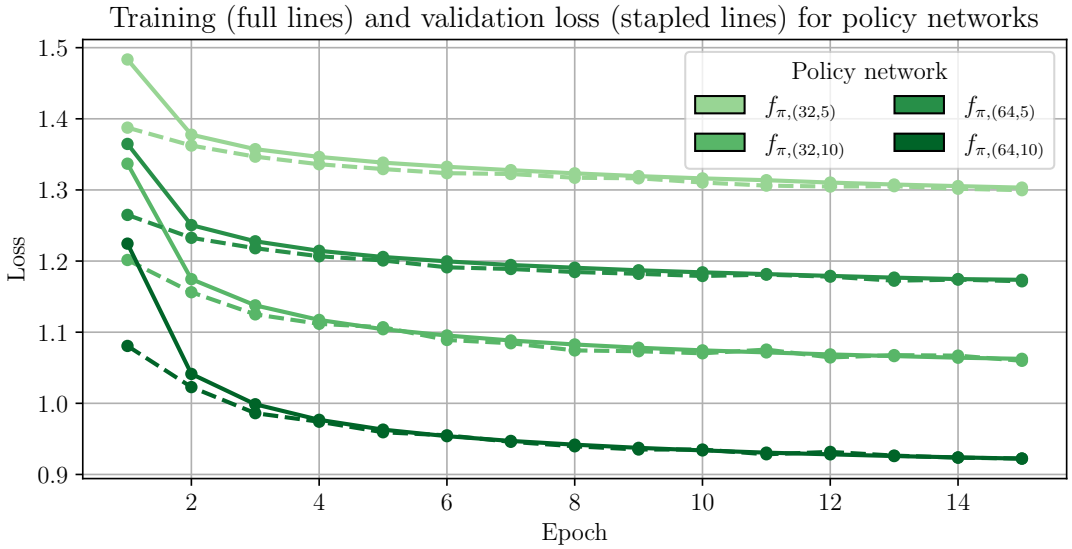


Figure 6.4: Training loss, $\mathcal{L}_\pi = \mathcal{L}_{\text{CE}} + \mathcal{R}$, and validation loss, $\mathcal{L}_{\text{CE}}$, over training epochs for each of the four policy network architectures with width $w \in \{32, 64\}$ and depth $d \in \{5, 10\}$. Training losses are full lines, validation losses are stapled lines.

To obtain a better interpretation of how accurate the policy networks are, we calculate Top-1 and Top-3 accuracy metrics on the validation set. Recall from (3.12) in Section 3.2.3 that this involves calculating the frequency of which the correct action is in the Top-1 and Top-3 actions with highest probability assigned by the policy network, based on the validation data. The results are shown in the left and right displays of Figure 6.5, respectively. The best-performing network achieved a Top-1 validation accuracy of around 0.66 and a Top-3 accuracy above 0.90. This indicates that the networks are able to learn patterns from the data, each classifying the correct action over 50% of the time.

Finally, we investigate the calibration of the policy networks. This is done by comparing the Top-1 accuracy and the Top-1 probabilities, split on the number of remaining moves. Top-1 probability is the highest probability assigned to any action on a given board by the policy network. Calibration can be interpreted as the networks having an appropriate level of "confidence", meaning that the probabilities they assign to the Top-1 action matches well with how often their predictions are correct.

The calibration plot for the policy network with $w = 64$ and $d = 10$ is presented in Figure 6.6. Here, we observe that the Top-1 accuracy follows the distribution of Top-1 probabilities quite closely, overlapping with the mean probability for all $T_i \leq 15$. This indicates that the network is properly calibrated: on average, it neither overshoots nor undershoots the probability of its prediction being correct. The Top-1 metrics are lower for higher T_i, which makes sense, since there are more actions to choose from. We also notice a bimodal pattern in the distribution of Top-1 probabilities for the lower move counts, especially for $T_i = 3$. This is likely caused by there being a duality among the possible solutions for states when few moves remain: either there is one correct action,

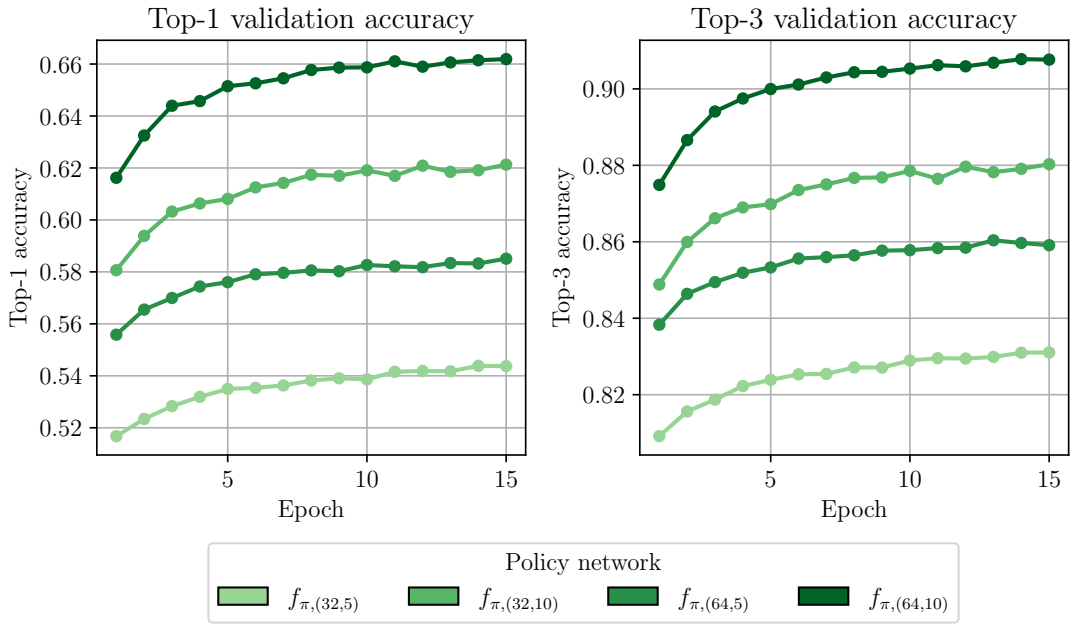


Figure 6.5: Top-1 (left) and Top-3 (right) accuracy on the validation set, for each of the four policy networks with width $w \in \{32, 64\}$ and depth $d \in \{5, 10\}$.

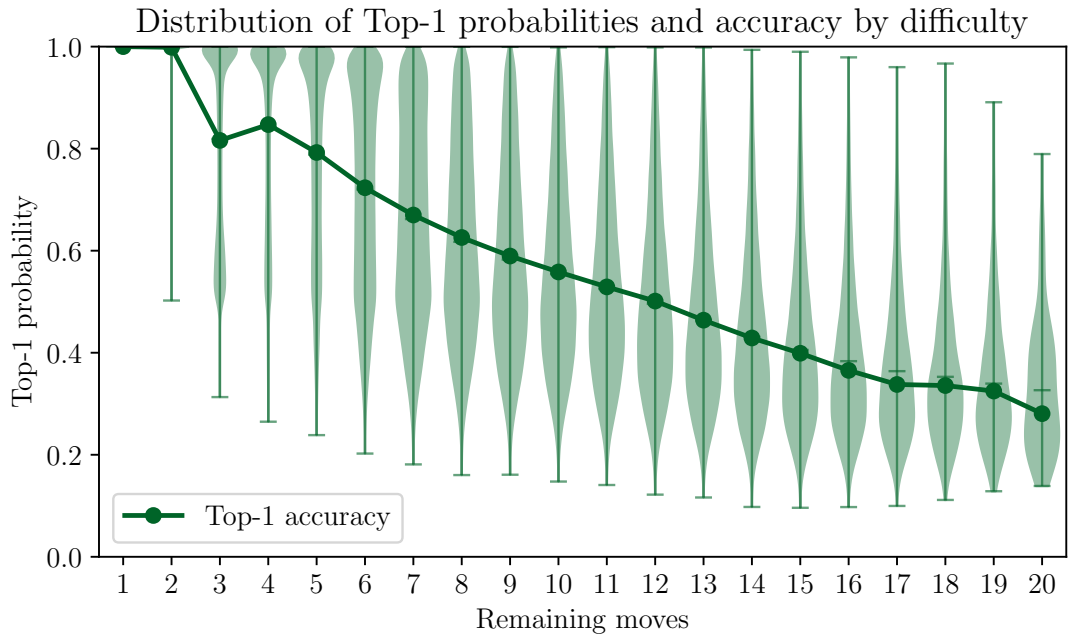


Figure 6.6: Distributions of Top-1 probabilities (violins) and Top-1 accuracy (green line) by board difficulty (remaining moves until the board is cleared). This is based on the policy network with $w = 64$ and $d = 10$. Top-1 probability is the highest probability assigned by the policy network to an action on a board, and Top-1 accuracy is the frequency of correct predictions. The center horizontal lines on the violins are the mean Top-1 probabilities.

in which case the network assigns a high probability to that action, or there are two or three equally good actions, in which case the probabilities are spread out, with one action being assigned a slightly higher probability. Although we do not show the plots here, the other networks gave similar results.

Next, we monitor the training and validation of the four value networks. We start by plotting losses over epochs, shown in Figure 6.7. The training losses in the left display seem to have converged sufficiently for our purpose, for all four networks. Validation losses converge in a somewhat jagged manner, but none increase by a substantial amount for higher epochs, indicating that we did not overfit to the training data. For the remainder of this chapter, we use the four value networks associated with the final epoch.

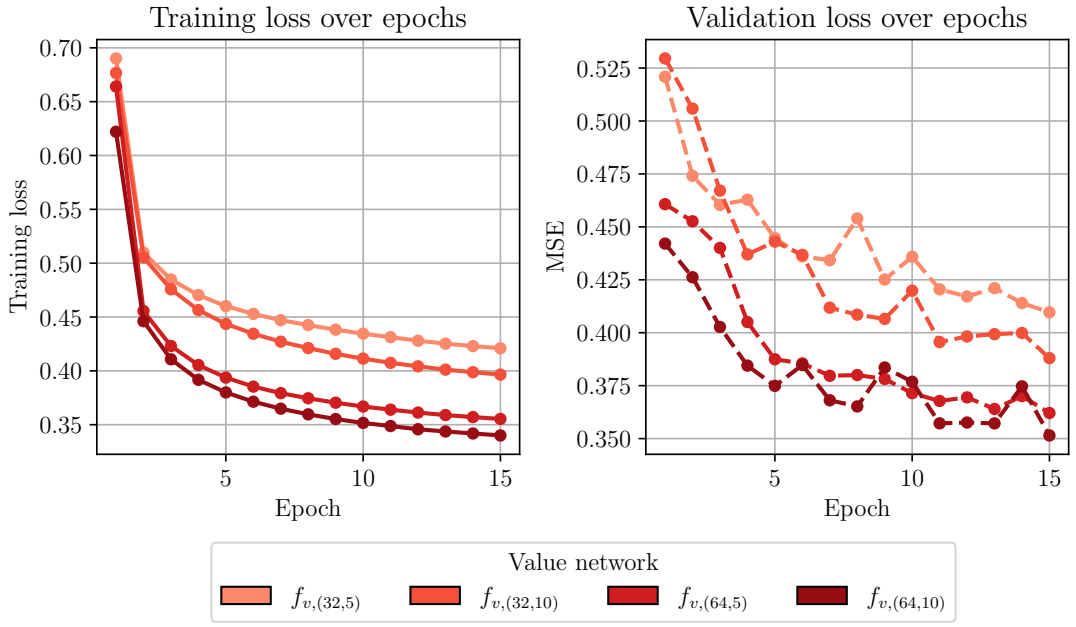


Figure 6.7: Training loss, $\mathcal{L}_v = \mathcal{L}_{\text{MSE}} + \mathcal{R}$, and validation loss, $\mathcal{L}_{\text{MSE}}$, over training epochs for each of the four value networks.

We analyze the precision of the value network with $d = 10$ and $w = 64$ by plotting the distributions of predicted number of remaining moves against the actual number for all samples in the validation data. The results are shown in Figure 6.8. Here, the orange line highlights where the actual number of moves used by beam search equals the predicted remaining number of moves, hence a perfectly accurate model would always lie on the orange line. We observe that the network is more accurate in its predictions the fewer moves that actually remain, which is as expected since there is less room for error. The tails of the distributions are heavier the more difficult the board is, and there is some overlap between the distributions for the higher moves. In particular, for difficult boards, the predictions typically lie within 1 to 2 moves from the mean, but up to 4 moves in the worst cases. Still, the means increase steadily, indicating that the network on average distinguishes between boards of different difficulties. Note also that the network undershoots the number of moves remaining for the most difficult boards. This is likely due to the data being slightly biased towards a lower number of remaining moves, as we observed from the distribution of move counts in Figure 5.3.

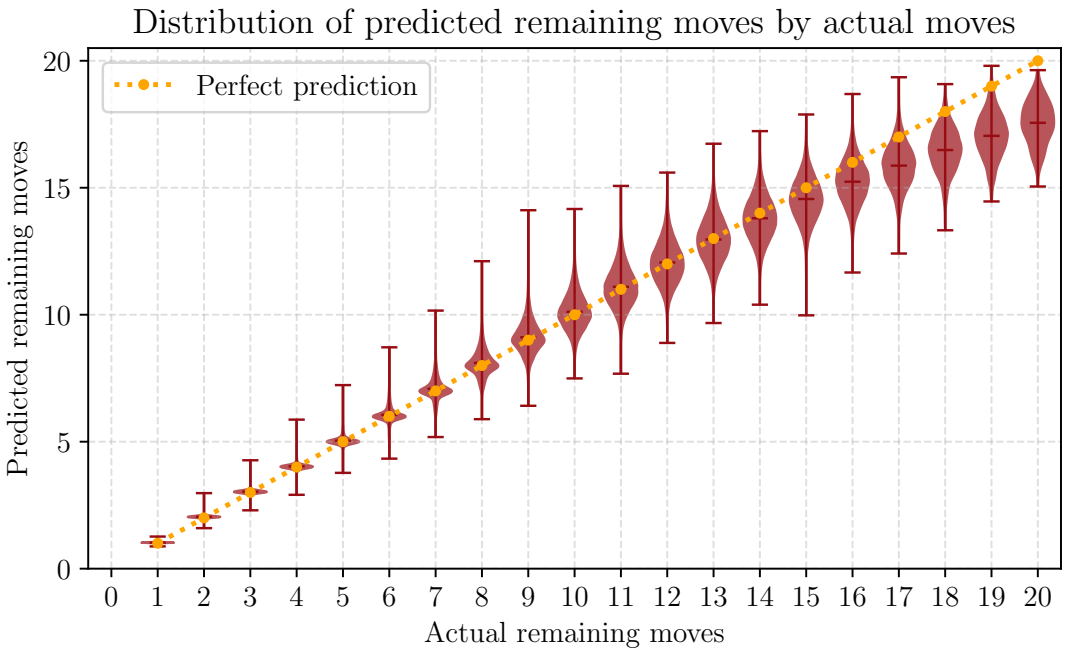


Figure 6.8: Distributions of predicted remaining number of moves (violins) versus the actual number of remaining moves, based on validation data. Predictions are made by the value network with $w = 64$ and $d = 10$. The orange line highlights where prediction equals the actual remaining moves in the dataset generated by beam search.

6.2.3 Evaluation

All eight networks were evaluated on the same 1 000 randomly generated boards as the self-made heuristics in Section 6.1, that is, using `np.random.randint` with `np.random.seed(22)`. Evaluation was performed through greedy play with the networks as heuristics. For value networks, this means that we use f_v to estimate the number of moves remaining after performing each action, and choose the action leading to the lowest prediction. For policy networks, we choose the action with maximum probability assigned by f_π. As in Section 6.1, the number of moves used and the time taken per board per network were recorded.

Aggregate results are presented in Table 6.4, and full distributions of move counts are shown in Figure 6.9. From these, we notice that the value networks on average perform better than policy networks, but at the cost of a higher runtime. This is as expected, considering that greedy play with value networks involve looking one step ahead, which requires more network evaluations but leads to better performance on its own. We also observe from Table 6.4 that the standard deviations are lower for policy networks than the value ones, and that the worst-case, maximum move counts are lower. This likely occurs because of the inaccuracy of value networks on more difficult boards, which we observed in Figure 6.8. It is not uncommon for value networks to predict move counts that deviate by 1 to 2 moves from the mean, which in the worst case can cause a repeating series of difficult boards: if many moves remain and the value network suggests the wrong action, the next board is essentially as difficult as the previous, which may lead to another poor choice of action, and so on. In the worst cases, this leads to a large deviation from the best-known solution. If the value network is quick to escape the difficult boards, however, it is accurate on the easier boards, and more likely finds the correct choices of actions.

Table 6.4: Aggregate performance of neural networks from greedy play on 1 000 randomly generated boards.

Policy network performance			
Model	Mean moves	Std. Dev.	Time per board (ms)
$f_{\pi,(32,5)}$	16.25	1.83	6.2
$f_{\pi,(32,10)}$	15.87	1.73	9.8
$f_{\pi,(64,5)}$	15.97	1.77	8.6
$f_{\pi,(64,10)}$	15.73	1.76	12.5
Value network performance			
Model	Mean moves	Std. Dev.	Time per board (ms)
$f_{v,(32,5)}$	15.57	1.87	101
$f_{v,(32,10)}$	15.56	1.96	179
$f_{v,(64,5)}$	15.35	1.85	149
$f_{v,(64,10)}$	15.34	1.92	285

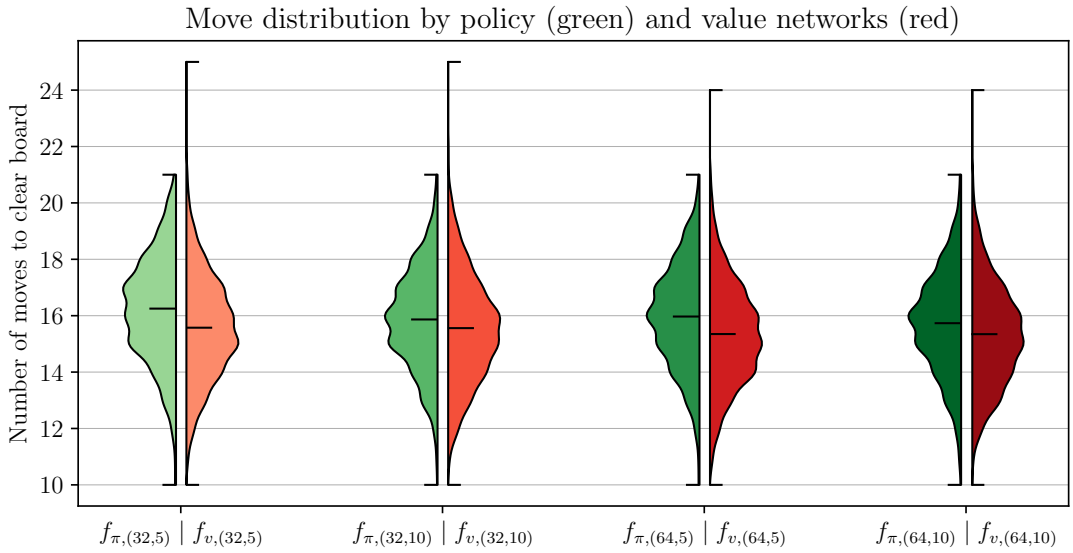


Figure 6.9: Distribution of move counts from greedy gameplay using policy and value networks, on 1 000 randomly generated *Former* boards. Green distributions belong to policy networks, and red distributions belong to value networks.

Compared to self-made heuristics, Table 6.4 shows that neural networks performed better when used in greedy play. All networks used substantially fewer moves than the 1 look-ahead heuristic, which we observed from Table 6.1 that on average used 18.54 moves to clear a board. All but the lightest policy network also beat the 2 look-ahead heuristic, which used 16.21 moves. The two value networks with $w = 64$ filters per layer even outperformed the 3 look-ahead heuristic, which used 15.48 moves on average and more than 10 times the runtime. This shows that the networks successfully learned a better strategy by imitating beam search with supervised learning.

We further analyze the policy and value networks through greedy play on the 100 daily boards. Again, the purpose of the models is not to solve the boards correctly, as they are meant to guide search techniques by being accurate and fast at board evaluations. Still, this comparison gives an indication as to how the networks perform on their own, which is relevant for how they might perform when used in a solver. As with the self-made heuristic models, we display the number of boards per difference to the best-known solutions, which are shown in Table 6.5. The policy network with $w = 64$ and $d = 10$ performed the best among the policy models, solving 8 out of 100 boards correctly. Policy networks were outperformed by value networks, the best of which being the ones with $w = 64$, solving 20 boards with greedy play. As discussed, this is to be expected considering that the greedy play with value networks involves a look-ahead strategy, at the cost of longer runtime.

Table 6.5: The number of boards per solution found by each supervised learning model through greedy play, measured in difference to the best-known solution, Δ.

Number of solutions per Δ (supervised learning models)						
Model	$\Delta = 0$	$\Delta = 1$	$\Delta = 2$	$\Delta = 3$	$\Delta = 4$	$\Delta \geq 5$
$f_{\pi,(32,5)}$	5	16	31	27	14	7
$f_{\pi,(32,10)}$	6	24	36	23	8	3
$f_{\pi,(64,5)}$	5	21	32	25	13	4
$f_{\pi,(64,10)}$	8	30	34	21	6	1
$f_{v,(32,5)}$	14	34	28	13	7	4
$f_{v,(32,10)}$	12	32	29	16	4	7
$f_{v,(64,5)}$	20	45	17	11	7	0
$f_{v,(64,10)}$	20	43	20	14	1	1

6.3 Proximal Policy Optimization

This section presents the results of using PPO to train reinforcement learning agents. We first monitor the training of the PPO algorithm, before we evaluate the actor and critic networks through greedy play on randomly generated and daily boards.

6.3.1 Training

To monitor the training process, we plot the reward obtained by the PPO models over time. This is shown in Figure 6.10, along with a moving average of both reward curves. Here, we observe that the average reward per episode $\bar{R}$ is initially as low as $\bar{R} = -27$,

which coheres well with random gameplay (Table 6.1). The rewards obtained over the course of both training procedures steadily increase over time. After 4 days of training, the average reward has stabilized just above $\bar{R} = -16$ for both models, meaning that the actor networks use less than 16 moves to clear a board on average. The smaller model converges faster due to quicker evaluations, which allows it to play more games in a shorter period of time than the larger model. In return, the larger model eventually reach a slightly higher mean reward.

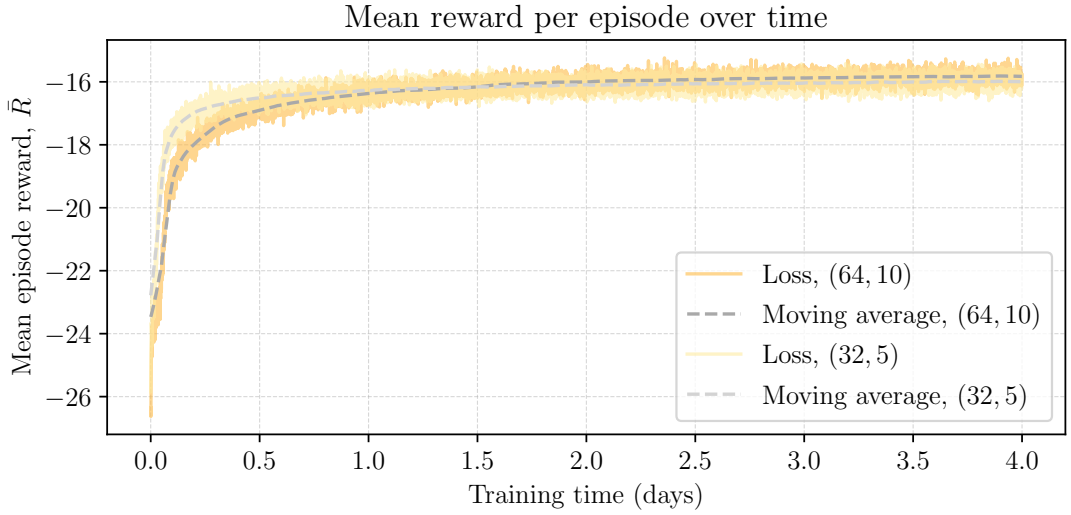


Figure 6.10: Mean episode reward, $\bar{R}$, over training time for both PPO training procedures. The stapled lines are moving averages of the reward curves, calculated using a 1 hour window centered at the respective time step.

Next, we plot each component in the PPO objective function defined in (3.23), that is, the actor loss, the critic loss and the negative entropy bonus, for both training procedures over time (Figure 6.11). We observe that there are fluctuations in the actor losses, but that these over time stabilize around zero. The critic losses decrease and flatten out over time, which is a sign of convergence. The entropy bonuses increase rapidly at the start, which is because the actors to begin with suggest actions quite randomly, and thus attain a high entropy. As the models learn how to play the game, it becomes more certain in its actions, thus more deterministic the probability distribution and the lower the entropy bonus. Since we set the entropy coefficient $c_{\text{ent}} = 0$, the negative bonus is not part of the training objective, and hence it does not decrease as time passes. However, the entropy bonuses not being very close to 0 indicate that the actor models encourage some exploration despite not being optimized to do so.

Finally, we check the calibration of the critics with respect to the actors. This is done by playing 10 000 games greedily with each actor, and for every state in every game, we store the prediction of the corresponding critic and the number of moves remaining based on the outcome of the game. This allows us to monitor how accurately each critic predicts the performance of the corresponding actor. We plot the distributions of the predictions for each number of remaining moves less than or equal to 20 for the $(64, 10)$ critic in Figure 6.12. The smaller critic gave similar results. Here, the orange line represents where a perfectly calibrated and unbiased actor would predict, that is, the line where prediction equals the actual outcome. The displayed results indicate that the critic is very accurate when few moves remain, and becomes less and less accurate as the number

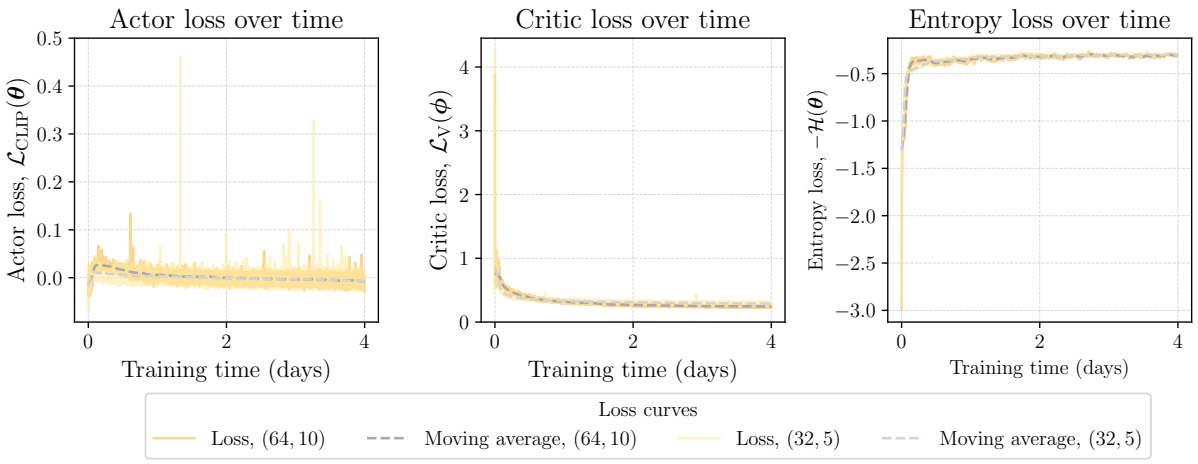


Figure 6.11: Actor loss (left), critic loss (middle), and negative entropy bonus (right), over training time, for networks with $(w, d) \in \{(32, 5), (64, 10)\}$.

of moves increases. This is similar to the value network accuracy in Figure 6.8. We also notice that the critic systematically predicts a lower number of moves remaining than what the actor actually uses, especially when more moves remain, which is remarkably similar to the pattern in Figure 6.8. This is likely caused by two things: Firstly, like in Figure 6.8, when the model generates data by playing many games, the collected data will be more biased towards easier boards, since these appear in all games. Secondly, which we hypothesize is the reason why the deviation is even greater for the PPO critic, is that the temporal difference factor $\lambda = 0.95$ is less than one, as we discussed in Section 6.3.1. Because of this, during training, when we minimize the MSE against $\hat{R}_t$ defined in (3.19), we minimize against estimates that are slightly biased towards lower move counts, leading to an even larger bias in the predictions of the critic.

6.3.2 Evaluation

We use the same procedure for evaluating the PPO-based models as we did for the supervised learning-based ones. That is, we play greedily with the two actors and critics on the two datasets, and monitor the number of moves per board and time used. The aggregate results from gameplay on the 1 000 randomly generated boards are presented in Table 6.6, and the full distributions of move counts are shown in Figure 6.13. The $(64, 10)$ actor used fewer moves on average than the $(32, 5)$ actor, although the latter performed better in the best and worst cases. This might just be up to random noise. The large actor used 15.76 moves on average to clear a board, which is better than the look-ahead heuristics (Table 6.1) and three of the supervised learning-based policy networks (Table 6.4). It is, however, slightly worse than the supervised learning-based model of same size. The smaller actor used 15.95 moves on average, thus performing better than the corresponding policy network trained with supervised learning. The runtime is approximately the same as for the supervised learning-based models, albeit slightly longer for the largest PPO actor. Greedy play with the $(32, 5)$ and $(64, 10)$ critics used 15.26 and 15.11 moves on average, respectively, which is substantially lower than any of the heuristics and value networks, in addition to having a lower runtime than the policy networks of the same respective sizes. The maximum number of moves used by the large critic is also lower than any other network or heuristic, using at most 20 moves to solve a board. Their run-

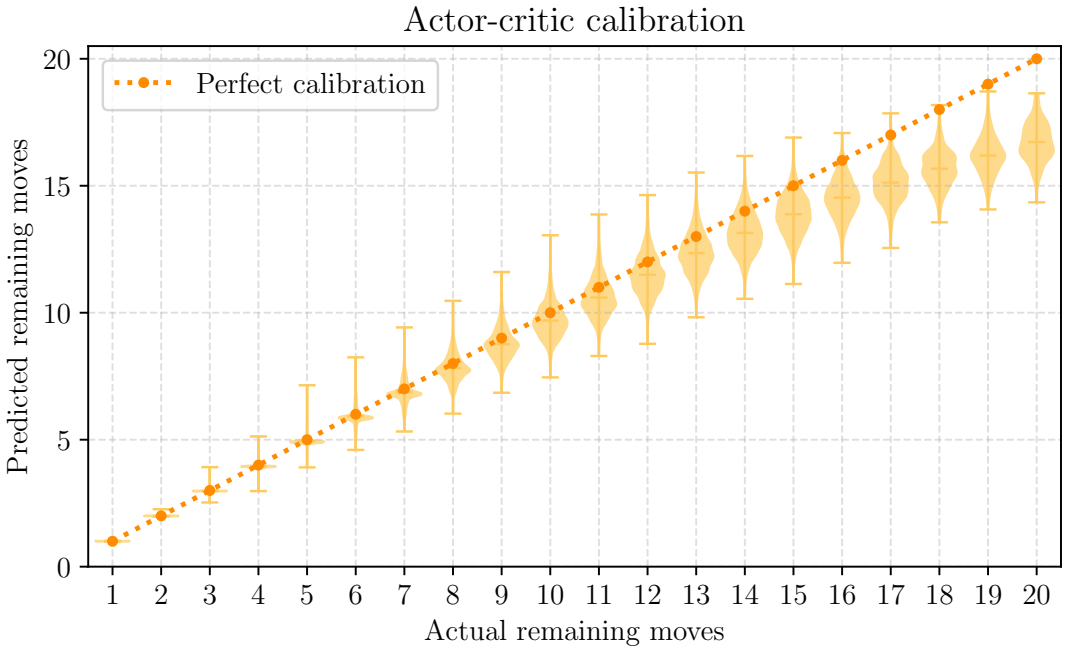


Figure 6.12: Actor-critic calibration for the PPO model with $w = 64$ and $d = 10$. Calibration shows the distribution of predicted moves by the critic per actual remaining number of moves, obtained through greedy play with the actor on 10 000 boards. The orange line is where the predicted remaining moves equals the actual remaining moves from greedy gameplay with the actor.

times also improve on that of the supervised learning models of similar sizes, and hence we would expect these models to perform well when combined with search techniques.

Table 6.6: Aggregate performance of the PPO actors and critics from greedy play on 1 000 randomly generated boards.

PPO actor and critic performance			
Model	Mean moves	Std. Dev.	Time per board (ms)
$\pi_{\boldsymbol{\theta},(32,5)}$	15.95	1.80	6
$\pi_{\boldsymbol{\theta},(64,10)}$	15.76	1.79	17
$v_{\boldsymbol{\phi},(32,5)}$	15.26	1.82	63
$v_{\boldsymbol{\phi},(64,10)}$	15.11	1.73	235

Next, we evaluate the performance of the PPO actors and critics on the set of 100 daily boards by comparing greedy play results to the best-known solutions. Recall that, although the purpose of these models is not to perfectly solve boards, the deviance from the best solution is a measure of accuracy and thus an indication of how they will perform with search techniques. As in the previous sections, the number of boards per difference is recorded, which we show in Table 6.7. Once again, the value models perform better than the policy models, which is due to the look-ahead strategy involved in greedy play. The small and large critics solved 19 and 21 boards correctly, respectively, which is about the same as the performance of the best value networks shown in Table 6.5, at 20 boards. In the worst cases, however, the critics perform worse, solving 11 and 9 boards with

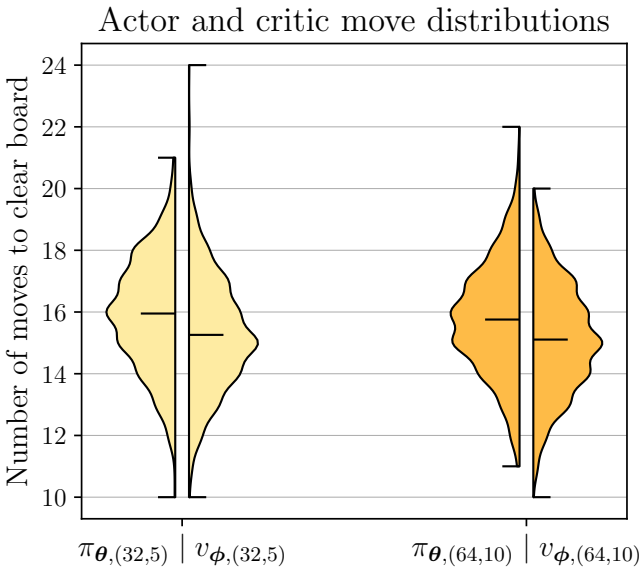


Figure 6.13: Distribution of move count for greedy gameplay with the actor and critic models trained with PPO, over 1 000 randomly generated *Former* boards. Each violin shows the distributions of move counts for the actor $\pi_{\boldsymbol{\theta},(w,d)}$ (left half) and the critic $v_{\boldsymbol{\phi},(w,d)}$ (right half) in each PPO network.

$\Delta \geq 5$, compared to 4 and 2 for the supervised learning-based value networks of the same sizes (Table 6.5). The same is also the case for the performances of the actors. The $(32, 5)$ actor solved 14 boards and the $(64, 10)$ actor solved 11 boards with $\Delta = 0$, which is a significant improvement from the 8 boards correctly solved by the best supervised learning-based policy network. But in the worst cases, the actors solved 14 and 9 boards with $\Delta \geq 5$, compared to 7 and 2 by the supervised learning-based policy networks. These results show that the PPO agents have learned strategies that are effective on some boards, but perhaps not as generalizable as those of the supervised learning models, leading to better performance in the best case and worse in the worst case.

Table 6.7: The number of boards per solution found by each PPO-based model, measured in difference to the best-known solution, Δ.

Number of solutions per Δ (PPO models)						
Model	$\Delta = 0$	$\Delta = 1$	$\Delta = 2$	$\Delta = 3$	$\Delta = 4$	$\Delta \geq 5$
$\pi_{\boldsymbol{\theta},(32,5)}$	14	22	23	17	10	14
$\pi_{\boldsymbol{\theta},(64,10)}$	11	26	23	15	16	9
$v_{\boldsymbol{\phi},(32,5)}$	19	31	13	15	11	11
$v_{\boldsymbol{\phi},(64,10)}$	21	27	23	13	7	9

6.4 Search Techniques

In this section, we present the results of using MCTS and beam search with self-made heuristics and neural networks to solve *Former* boards. Recall that self-made heuristics, supervised learning-based networks, and PPO-based networks are referred to as models,

which make predictions based on a given board. Once combined with search techniques, we refer to them as solvers, whose purpose is to find solutions that use as few moves as possible.

The section is split into two parts. First, we check the performance of each solver on the same 1 000 randomly generated boards we have used in the previous two sections. In the second part, we run each solver on the dataset containing 100 official *Former* boards. Since we have the best-known solution to each board, this allows us to truly check how close our solvers are to solving *Former*.

6.4.1 Performance on Random Boards

We run each MCTS and beam search solver on the same 1 000 randomly generated boards used in the previous sections. For each run, we set a time limit of $t_{\max} = 10$ seconds per board and record the time stamp and solution length for each solution found. Using this, we calculate the best-so-far solution, that is, the average best solution up to time t for $t \in [0.5, 10]$ seconds, per solver. We do not plot the best-so-far solution for $t < 0.5$ seconds since the first solutions found by each model typically are the easiest boards, with a lower than average move count, and thus the average solution does not represent the entire dataset for low values of t.

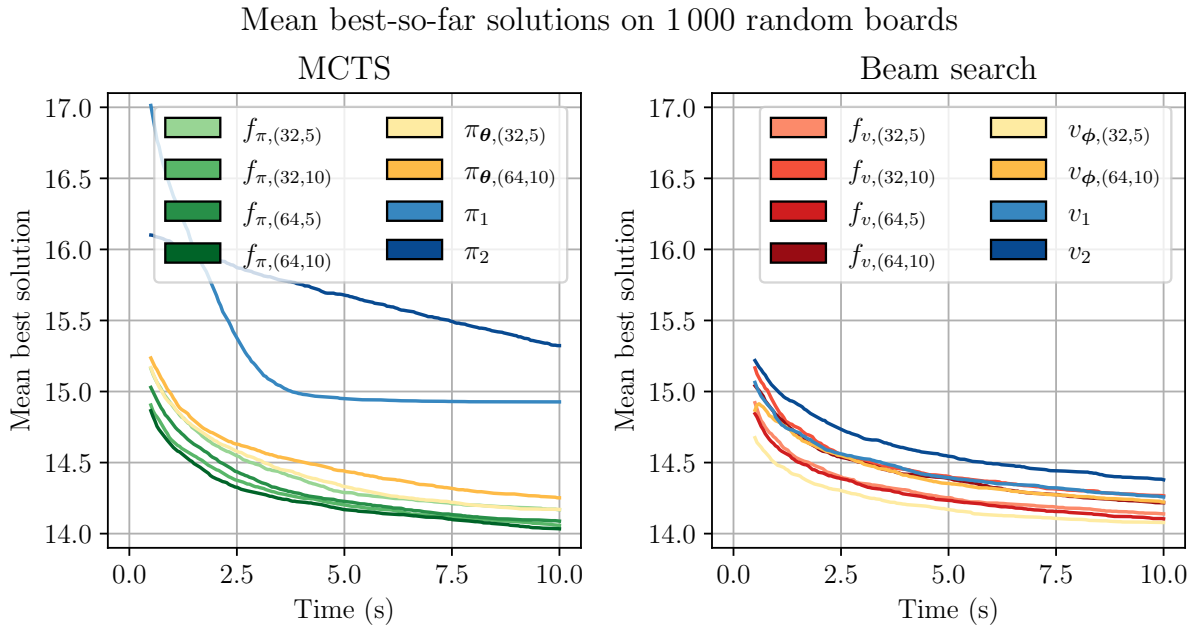


Figure 6.14: Average best solution found by each MCTS solver (left display) and beam search solver (right display) within a search time of $t \in [0.5, 10]$ seconds. Based on 1 000 randomly generated boards.

The resulting best-so-far curves are shown in Figure 6.14, and aggregate results after $t_{\max} = 10$ seconds of search are presented in Table 6.8. From the left and right displays of Figure 6.14, we observe that the solutions start relatively high and decline quickly as we let the search continue. The neural network-based solvers significantly outperform self-made heuristics, particularly when used with MCTS. The heuristics perform better with beam search, but also here, they are outperformed by the neural networks. This indicates that the networks indeed have picked up on patterns in *Former* boards, making

them better approximations of the optimal policy and value functions than the self-made heuristics. The largest PPO actor performed the worst among the network-based MCTS solvers, which makes sense considering that it was slower and not significantly more accurate than the other network models (Tables 6.4 and 6.6). What is more surprising considering the performance on its own, is the largest PPO critic, performing at the same level as the largest value network, $f_{v,(64,10)}$, despite being faster and more accurate in initial analysis. Overall, the best-performing solver combined MCTS with $f_{\pi,(64,10)}$, reaching 14.03 moves per board on average with a standard deviation of 1.42 after 10 seconds of search (Table 6.8). This is a significant improvement on the model on its own, which used 15.73 moves on average with a standard deviation of 1.76 in greedy play on the same set of boards (Table 6.4). Note that the standard deviation has an unknown lower bound determined by the variation in the optimal solutions to the boards, which are unknown.

Table 6.8: Aggregate results after $t_{\max} = 10$ seconds of search on 1 000 randomly generated boards with each MCTS and beam search solver.

MCTS solver performances		
Incorporated model	Mean moves	Std. Dev.
π_1	14.93	1.56
π_2	15.32	1.79
$f_{\pi,(32,5)}$	14.17	1.47
$f_{\pi,(32,10)}$	14.06	1.42
$f_{\pi,(64,5)}$	14.09	1.44
$f_{\pi,(64,10)}$	14.03	1.42
$\pi_{\boldsymbol{\theta},(32,5)}$	14.17	1.45
$\pi_{\boldsymbol{\theta},(64,10)}$	14.25	1.50
Beam search solver performances		
Beam search solver	Mean moves	Std. Dev.
v_1	14.26	1.53
v_2	14.38	1.61
$f_{\pi,(32,5)}$	14.14	1.56
$f_{\pi,(32,10)}$	14.27	1.62
$f_{\pi,(64,5)}$	14.10	1.54
$f_{\pi,(64,10)}$	14.22	1.59
$v_{\boldsymbol{\phi},(32,5)}$	14.08	1.48
$v_{\boldsymbol{\phi},(64,10)}$	14.22	1.56

6.4.2 Performance on Daily NRK Boards

In this section, we present the results of using MCTS and beam search solvers on the set of 100 daily boards from the official NRK website. The dates and respective boards are available through the GitHub repository, link in Appendix B.1. As mentioned, each of these boards has an associated best-known solution, which is the lowest score obtained by any player that day. The average of these solutions is 13.6 moves, with a standard

deviation of 1.36. We first show the average number of moves used by each solver, then we analyze how far the best solvers deviate from the best-known performance over time, before we display the total proportion of daily boards solved over time, across all solvers. Then, we analyze how many equivalent best solutions that exist to a couple handpicked boards, before attempting to find better solutions than what is known.

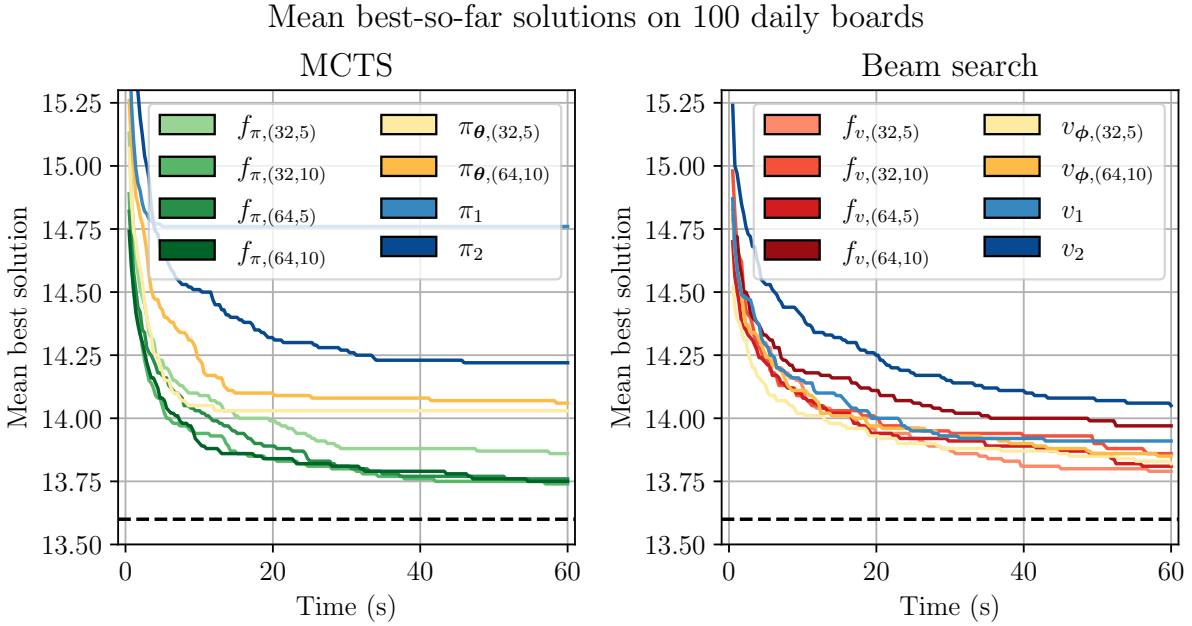


Figure 6.15: The average best-so-far solution over time for each MCTS and beam search model. The best-so-far solution at time t is the best solution found by the respective model up until time t. The black, stapled lines are the average number of moves used in the best solutions published by NRK, 13.6 moves.

The average best solution per time for the MCTS and beam search solvers are shown in the left and right displays of Figure 6.15, respectively. The black stapled lines at 13.6 moves indicate the average number of moves used by the best-known solutions. From these plots, we observe that MCTS with policy networks use fewer moves than the beam search solvers, reaching as low as 13.74 moves on average after 60 seconds. This is consistent with the results from evaluation on random boards. There is more randomness involved in which model performs the best over time compared to on the randomly generated boards, which is likely due to the dataset being smaller. For MCTS, the heavier policy networks typically perform better, with the exception of the largest PPO actor. Both PPO actors performed worse than the other MCTS solvers with network models, which may be due to the actors being deterministic, and that MCTS alone does not provide sufficient exploration. This would explain why the best-so-far curves flatten out relatively quickly. For beam search solvers, the lightest value network, $f_{v,(32,5)}$, performs the best, followed by $f_{v,(64,5)}$ and the PPO critic. This indicates that lighter networks provide sufficiently accurate predictions when combined with beam search. Overall, the observations from Figure 6.15 show that the neural network approaches outperform the self-made heuristics. This further confirms that the neural networks we train using supervised learning and PPO are able to learn spatial patterns from *Former* boards, and make reasonable predictions based on them.

Next, we monitor how far the solutions deviate from the best-known solutions over

time. We find the difference (Δ) between the number of moves used by our models and the best solutions, for each board, given a time constraint $t_{\max} \in \{1, 5, 10, 30, 60\}$. Table 6.9 presents the number of boards by Δ and $t_{\max}$, for the model that solved the most boards correctly per search technique. With a time constraint of 1 second, the best-performing beam search model solves 40 out of the 100 boards, beating the best-performing MCTS model, which solved 31 boards correctly. This is likely due to beam search being faster in exploring the top recommended actions, whereas MCTS explores each possible action from the initial state at least $N_{\min} = 10$ times before focusing on the most promising ones. For easier boards, where the value and policy networks approximate v^* and π^* well, such greedy search leads beam search to find the best solution faster than MCTS. For larger $t_{\max}$, however, MCTS models consistently outperform beam search models, both in terms of number of correct solutions and deviation from the best solution. Already after $t_{\max} = 5$ seconds, the MCTS models deviate by 2 moves in the worst case, whereas the beam search model misses the best solution by as much as 4 moves. This pattern is consistent with what we observed when analyzing the neural networks on their own in Section 6.2.3, where we found that the policy networks had a lower standard deviation, which indicates that they more consistently predict a good course of action. As a result, the solutions found by MCTS rarely deviate significantly from the best solutions, even for smaller time limits. After $t_{\max} = 60$ seconds, the best MCTS model solved 86 out of 100 boards correctly, 14 solutions deviated by 1 move, and not a single solution deviated by 2 moves. The best beam search model solved 83 boards correctly, 15 solutions deviated by 1, and 2 solutions deviated by 2.

Table 6.9: The number of boards solved by the best solver in a search using the same number of moves as the best solution ($\Delta = 0$), using one more move ($\Delta = 1$), and so on, for a selection of $t_{\max}$.

Number of solutions per Δ and $t_{\max}$ (MCTS)					
$t_{\max}$ (s)	$\Delta = 0$	$\Delta = 1$	$\Delta = 2$	$\Delta = 3$	$\Delta = 4$
1	31	50	14	5	0
5	58	36	6	0	0
10	72	26	2	0	0
30	80	20	0	0	0
60	86	14	0	0	0
Number of solutions per Δ and $t_{\max}$ (beam search)					
$t_{\max}$ (s)	$\Delta = 0$	$\Delta = 1$	$\Delta = 2$	$\Delta = 3$	$\Delta = 4$
1	40	40	12	4	4
5	55	33	9	1	1
10	62	34	4	0	0
30	77	19	4	0	0
60	83	15	2	0	0

In addition to observing the performance of the best solvers alone, we aggregated the results from several solvers over time. This allows us to check the proportion of the daily boards we are able to solve correctly, regardless of which solver found the solution. This is interesting to check since there is some randomness involved in whether a model

finds a solution or not, especially when using MCTS, which samples actions pseudo-randomly from the policy during simulations. Aggregating over several solvers reduces the variance caused by such randomness, which hence gives a slightly more balanced view of the performance of each search technique and the combined performance of our solvers. In Figure 6.16, we have aggregated the solutions found by MCTS solvers, the solutions found by beam search solvers, and the solutions found by all solvers combined. Based on this, we display the proportion of boards solved correctly as a function of time. Here we observe similar results to what we did in Table 6.9: Beam search performs better than MCTS during the first seconds, but is eventually overtaken by MCTS, which solves more boards accurately given a sufficient amount of time. Across all solvers, 73 of the 100 daily boards were solved in less than a second of search, and 91 were solved before 10 seconds had passed. After 60 seconds, the beam search models together solved 92 boards, MCTS models found the solution to 96 boards, and combined, all solvers found the best-known solutions to 98 out of 100 boards. This shows that modern machine learning techniques can be used to solve *Former*. The two boards we did not solve were the ones on February 15$^{\text{th}}$ and March 24$^{\text{th}}$. To check if we are able to find the solutions to these boards at all, we used MCTS with $f_{\pi,(64,10)}$ as guidance for 24 hours. With this extended time limit, we found the best-known solution to the board on February 15$^{\text{th}}$ in 10 minutes, but we never found the solution to the board on March 24$^{\text{th}}$, even after 24 hours of search. This demonstrates that, while our solvers typically find the best-known solution within a minute (Figure 6.16), they do not guarantee success on every board.

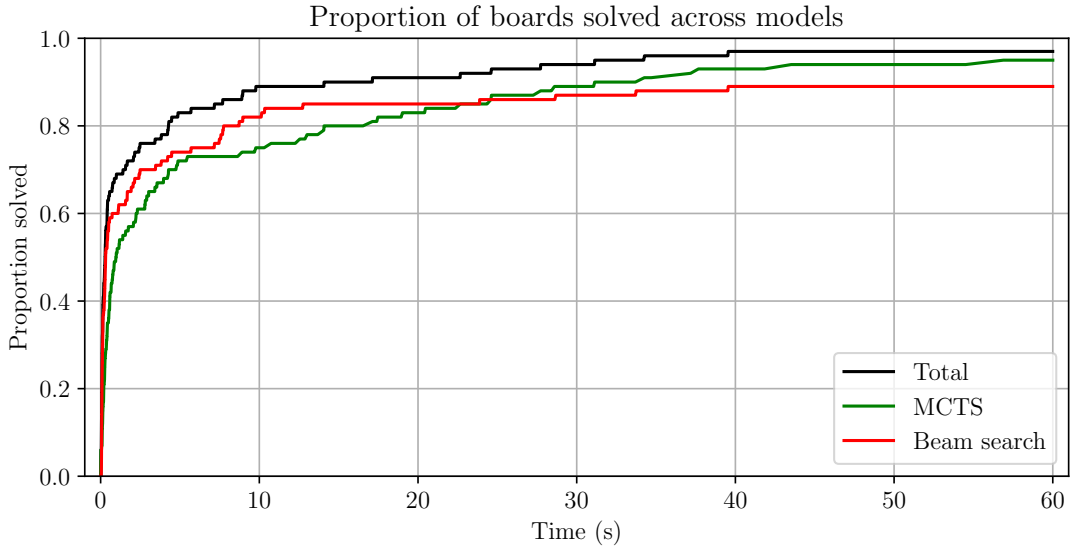


Figure 6.16: Proportion of 100 daily boards solved across models for each search technique separately, and aggregated across both.

Now that we have analyzed the overall performance of solvers across all boards, we switch focus to a few hand-picked boards, and investigate how many equivalent best solutions that exist to them. For each board, we run the MCTS solver with $f_{\pi,(64,10)}$ as guidance until it finds the best-known solution, 1 000 times, and record the initial action of each solution along with the total number of possible actions from that board. This allows us to monitor the proportion of actions that can be chosen to obtain the best-known solution from a given state. We refer to one such action as an *optimal action*. After doing this from the initial board, we perform the optimal action that was chosen

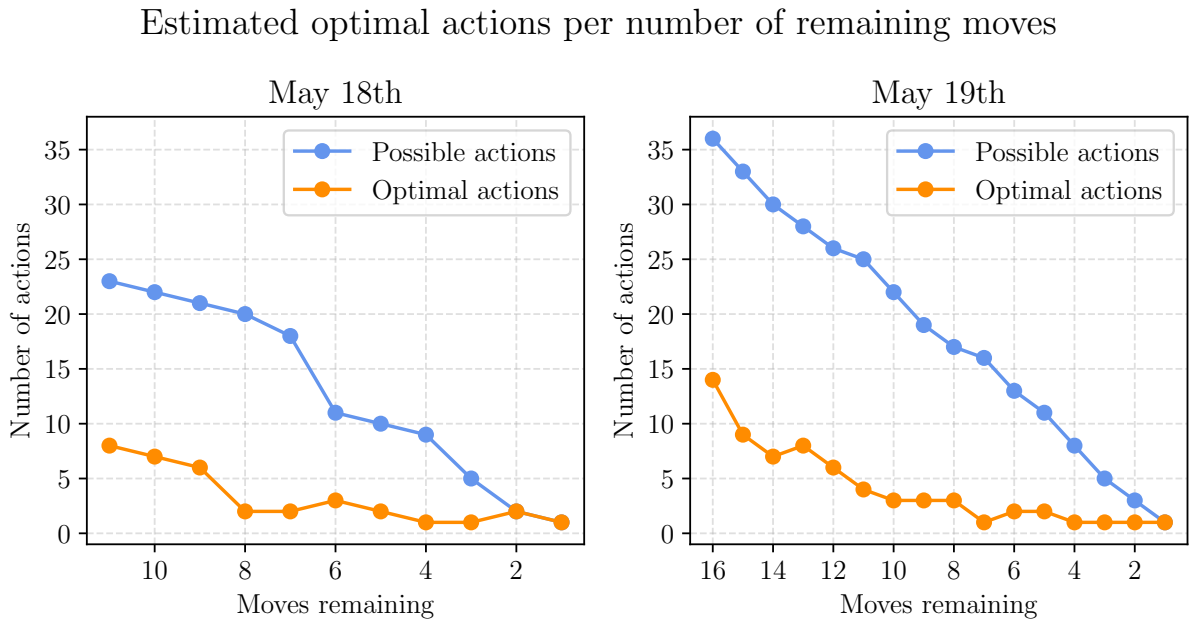


Figure 6.17: The number of possible actions and the number of optimal actions (actions that lead to a best-known solution) per number of remaining moves for the boards published on May 18$^{\text{th}}$ (left display) and May 19$^{\text{th}}$ (right display). The number of optimal actions per number of moves remaining is based on 1 000 solutions found using the MCTS solver with $f_{\pi,(64,10)}$ as guidance, along the path of the most frequently found solution.

most frequently among the 1 000 solutions, and find 1 000 new solutions using the lowest known number of moves from the next board, with the same procedure. By repeating this for each action along the most "popular" best path, we find a rough estimate of the number of optimal actions per remaining number of moves. We do this for the boards published by NRK on May 18$^{\text{th}}$ and 19$^{\text{th}}$, which have varying difficulties: the first can be solved in 11 moves, and the second has 16 as the best-known solution. The number of optimal and possible actions per number of remaining moves for the two boards is shown in the left and right displays of Figure 6.17. These plots show that surprisingly large amounts of initial actions can be taken to obtain the best-known solution: on May 18$^{\text{th}}$, 8 out of 23 moves were optimal, and on May 19$^{\text{th}}$, 14 out of 36. This is likely due to there being several choices of initial actions that lead to the same board, as discussed in Section 2.2.2. For both boards, the relative number of optimal actions decreases rapidly. When 8 moves remain, only 2 out of 20 actions lead to a best solution for the board on May 18$^{\text{th}}$, and for the board on May 19$^{\text{th}}$, when 7 moves remain, only 1 out of 16 moves were optimal. This indicates that the middle part of the game is the most crucial, as there perhaps no longer exist several combinations of actions that lead to the same outcome, hence it is critical to choose the correct action in order to obtain the best-known solution. In the final parts of the games, we observe that there are typically 1-2 actions that are optimal. This supports the hypothesis we made on the bimodal pattern in the policy network accuracy in Figure 6.6: typically, the network only has 1 or 2 actions when few moves remain, and hence it either assigns a probability close to 1 or close to 0.5 to the correct actions.

A common question we were asked while working on this project was: "do you ever beat the best-known solution?" The answer to this is yes, if we are sufficiently early at

work. In our experience, the best solution found among any player on a given day is typically found within a few hours after the board has been released, which either is due to there being very many players, some very good players, or because there are more people like us, using machine learning or other techniques to solve boards quickly. If we employ our models soon after the release of a new board, however, we are typically the first to set the best record that day.

Another frequently asked question is related to whether we can be certain that the best-known solutions are actually the best possible solutions. We cannot be certain of this without actually searching the entire state space, which is in practice impossible. However, as a test, we ran MCTS with the $f_{\pi,(64,10)}$ network as guidance for 24 hours on four daily boards: May 17$^{\text{th}}$, May 18$^{\text{th}}$, May 19$^{\text{th}}$, and May 20$^{\text{th}}$, with best-known solutions of 13, 11, 16 and 13 moves, respectively. On these boards, MCTS found the best-known solutions in approximately 6, 0.1, 12 and 3 seconds, but did not improve in the remainder of the 24 hours. Considering that MCTS finds the best-known solutions in only a few seconds but never improves on them in 24 hours of search, it is reasonable to assume that the best-known solutions to these boards likely are the best possible solutions.

CHAPTER

SEVEN

CONCLUSIONS

7.1 Concluding Remarks

The purpose of this thesis was to solve the single-player puzzle *Former* by NRK (2024) with machine learning techniques, and our initial goal was to solve boards in less than a minute of searching. To do so, we formulated *Former* as a Markov decision process, and defined a mathematical solution as finding the optimal policy function π^* or the optimal value function v^*. In *Former*, π^* corresponds to the strategy that always clears the board in the fewest number of moves possible, and v^* corresponds to the number of moves remaining if the player acts according to π^*. These two functions were approximated by the use of three distinct approaches: by crafting self-made heuristics, by training policy and value networks on self-generated data, and by using Proximal Policy Optimization (PPO), a state-of-the-art reinforcement learning technique. In greedy play, where we at each step choose the best action according to model predictions, the neural network-based models significantly outperformed the self-made heuristics. The largest PPO critic performed the best, on average solving *Former* boards in 15.11 moves through greedy play.

We also formulated solving *Former* as a search problem, and used Monte Carlo Tree Search (MCTS) and beam search guided by the approximated policy and value functions to search for solutions to a variety of boards. Our findings show that modern machine learning methods combined with search techniques efficiently solve most *Former* boards, in total finding the best solution to 73% of boards in less than a second and 98% in less than a minute of searching, thus exceeding our initial goal. Neural networks obtained from supervised learning and PPO significantly outperformed any self-made heuristic we crafted, showing the efficiency of neural networks to approximate optimal policy and value functions. Among the individual solvers, MCTS with policy networks as guidance generally performed the best, with the policy network consisting of 10 layers and 64 filters per layer being the best at 14.03 moves on average to clear a board. On the daily boards, all top 3 solvers combined MCTS with policy networks, solving up to 86 out of 100 boards by themselves in less than a minute.

7.2 Future Work

Throughout the work on this thesis, we experimented with several different approaches to solving *Former*, many of which did not work. Some of these, such as brute-force search, are simply unfeasible due to the large state space, but other methods may work given enough time and efficient implementation. These are prospective methods for future work. In particular, we tried using an AlphaZero-inspired approach (Silver et al., 2018), where MCTS is used as a *planner*, that plays thousands of games, generating probability distributions over actions based on each search. These distributions are in turn used to update the weights of a dual-head network, which again is used to generate more data, in a typical model-based reinforcement learning manner. The main caveat of this method is that it requires training for a very long time, which hindered us from attaining the results we were hoping for. However, given efficient coding, sufficient computational power, and more time at hand, we believe that this approach may outperform what we have achieved in this thesis.

Another topic for future work is to expand on our current methodology. Due to the limited resources devoted to this work, we did not spend a significant amount of time tuning the parameters in PPO, testing more network architectures in the supervised learning setting, or attempting other methods of generating data. If done correctly, such fine-tuning of our methods could lead to improved performance on *Former* boards, perhaps solving all 100 boards in less than a minute, and not just 98. Additionally, in this thesis, our focus was on the use of MCTS and beam search as search techniques, but there are many other alternatives that may perform well given the proper models to guide them. One idea is to use classical pathfinding algorithms such as A* (Hart et al., 1968) and IDA* (Korf, 1985), and another option is to use swarm-intelligence techniques such as Ant Colony Optimization (Dorigo et al., 2007) and the Grey Wolf Optimizer (Mirjalili et al., 2014). Each of these could be combined with our neural network models and may prove efficient for the problem of solving *Former*.

As an alternative next step, the methods developed in this thesis are applicable to other games as well. Although many well-known games have already been tried and solved by similar methods, new puzzles emerge quite frequently, most of which inherit the same competitive and strategic aspects as *Former*, making them prime targets for modern machine learning techniques.

REFERENCES

Agostinelli, F., McAleer, S., Shmakov, A., & Baldi, P. (2019). Solving the rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, *1*(8), 356–363. https://doi.org/10.1038/s42256-019-0070-z

Altmann, A., Toloşi, L., Sander, O., & Lengauer, T. (2010). Permutation importance: A corrected feature importance measure. *Bioinformatics*, *26*(10), 1340–1347. https://doi.org/10.1093/bioinformatics/btq134

Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for hyper-parameter optimization. *Advances in Neural Information Processing Systems*, *24*, 2546–2554. https://papers.nips.cc/paper_files/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf

Berner, C., Brockman, G., Chan, B., Cheung, V., Cheung, J., Dennison, L., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., et al. (2019). Dota 2 with large scale deep reinforcement learning. *Proceedings of the 2019 International Conference on Machine Learning*. https://doi.org/10.48550/arXiv.1912.06680

Bisiani, R. (1987). Beam search. In S. C. Shapiro (Ed.), *Encyclopedia of artificial intelligence* (pp. 56–58). John Wiley & Sons.

Breiman, L. (2001). Random forests. *Machine learning*, *45*, 5–32. https://doi.org/10.1023/A:1010933404324

Brochu, E., Cora, V. M., & De Freitas, N. (2010). A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning. *arXiv*. https://doi.org/10.48550/arXiv.1012.2599

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., & Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, *4*(1), 1–43. https://doi.org/10.1109/TCIAIG.2012.2186810

Cliff, A. D., & Ord, J. K. (1981). *Spatial processes : Models & applications*. Pion.

Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. *International conference on computers and games*, 72–83. https://doi.org/10.1007/978-3-540-75538-8_7

Cover, T. M., & Thomas, J. A. (2005). Entropy, relative entropy, and mutual information. In *Elements of information theory* (pp. 13–55). John Wiley & Sons. https://doi.org/10.1002/0471200611.ch2

Dorigo, M., Birattari, M., & Stutzle, T. (2007). Ant colony optimization. *IEEE computational intelligence magazine*, *1*(4), 28–39. https://doi.org/10.1109/MCI.2006.329691

Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., R. Ruiz, F. J., Schrittwieser, J., Swirszcz, G., et al. (2022). Discov-

ering faster matrix multiplication algorithms with reinforcement learning. *Nature*, *610*(7930), 47–53. https://doi.org/10.1038/s41586-022-05172-4

Fisher, R. A. (1970). Statistical methods for research workers. In *Breakthroughs in statistics: Methodology and distribution* (pp. 66–70). Springer. https://doi.org/10.1007/978-1-4612-4380-9_6

Flatwhite Studios. (2024). *Spotle*. Retrieved June 7, 2025, from https://spotle.io/

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107. https://doi.org/10.1109/TSSC.1968.300136

Ho, J., & Ermon, S. (2016). Generative adversarial imitation learning. *Advances in neural information processing systems*, *29*. https://doi.org/10.48550/arXiv.1606.03476

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *International conference on machine learning*, 448–456. https://doi.org/10.48550/arXiv.1502.03167

Jais, I. K. M., Ismail, A. R., & Nisa, S. Q. (2019). Adam optimization algorithm for wide and deep neural network. *Knowl. Eng. Data Sci.*, *2*(1), 41–46. https://doi.org/10.17977/um018v2i12019p41-46

Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., et al. (2021). Highly accurate protein structure prediction with alphafold. *Nature*, *596*, 583–589. https://doi.org/10.1038/s41586-021-03819-2

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv*. https://doi.org/10.48550/arXiv.1412.6980

Kocsis, L., & Szepesvári, C. (2006). Bandit based monte-carlo planning. *European conference on machine learning*, 282–293. https://doi.org/10.1007/11871842_29

Konda, V., & Tsitsiklis, J. (1999). Actor-critic algorithms. *Advances in neural information processing systems*, *12*. https://papers.nips.cc/paper/1786-actor-critic-algorithms

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, *27*(1), 97–109. https://doi.org/10.1016/0004-3702(85)90084-0

Korf, R. E. (1997). Finding optimal solutions to rubik's cube using pattern databases. *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 700–705. http://www.aaai.org/Library/AAAI/1997/aaai97-109.php

Lin, M., Chen, Q., & Yan, S. (2013). Network in Network. *arXiv*. https://doi.org/10.48550/arXiv.1312.4400

LinkedIn. (2024). *Queens*. Retrieved June 7, 2025, from https://www.linkedin.com/games/queens

Mirjalili, S., Mirjalili, S. M., & Lewis, A. (2014). Grey wolf optimizer. *Advances in engineering software*, *69*, 46–61. https://doi.org/10.1016/j.advengsoft.2013.12.007

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *International conference on machine learning*, 1928–1937. https://doi.org/10.48550/arXiv.1602.01783

NRK. (2024). *Former* (M. Folkestad, K. A. Andersen, M. V. Feiring, R. Rognan, & J. S. Jensen, Eds.). Retrieved June 7, 2025, from https://www.nrk.no/spill/former

Odland, T. (2024). *Solving nrk's game former*. Retrieved May 14, 2025, from https://tommyodland.com/articles/2024/solving-nrks-game-former/

Ogundokun, R. O., Maskeliunas, R., Misra, S., & Damaševičius, R. (2022). Improved cnn based on batch normalization and adam optimizer. *International Conference on Computational Science and Its Applications*, 593–604. https://doi.org/10.1007/978-3-031-10548-7_43

O'Shea, K., & Nash, R. (2015). An introduction to convolutional neural networks. https://doi.org/10.48550/arXiv.1511.08458

Pearl, J., & Korf, R. E. (1987). Search techniques. *Annual Review of Computer Science*, *2*(1), 451–467. https://doi.org/10.1146/annurev.cs.02.060187.002315

Puterman, M. L. (1990). Markov Decision Processes. *Handbooks in Operations Research and Management Science*, *2*, 331–434. https://doi.org/10.1016/S0927-0507(05)80182-3

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021). Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, *22*(268), 1–8. https://github.com/DLR-RM/stable-baselines3

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. https://doi.org/10.1038/323533a0

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. (2015). Imagenet large scale visual recognition challenge. *International journal of computer vision*, *115*, 211–252. https://doi.org/10.1007/s11263-015-0816-y

Russell, S. J., & Norvig, P. (2016). *Artificial intelligence: A modern approach*. pearson.

Schadd, M. P. D., Winands, M. H. M., van den Herik, H. J., Chaslot, G. M. J. B., & Uiterwijk, J. W. H. M. (2008). Single-player monte-carlo tree search. *Computers and Games: 6th International Conference, CG 2008, Beijing, China, September 29-October 1, 2008. Proceedings 6*, 1–12. https://doi.org/10.1007/978-3-540-87608-3_1

Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2015). High-Dimensional Continuous Control Using Generalized Advantage Estimation. *arXiv*. https://doi.org/10.48550/arXiv.1506.02438

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *arXiv*. https://doi.org/10.48550/arXiv.1707.06347

Shannon, C. E. (1950). Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, *41*(314), 256–275. https://doi.org/10.1080/14786445008521796

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489. https://doi.org/10.1038/nature16961

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, *362*(6419), 1140–1144. https://doi.org/10.1126/science.aar6404

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *Nature*, *550*(7676), 354–359. https://doi.org/10.1038/nature24270

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, *25*. https://doi.org/10.48550/arXiv.1206.2944

Sutton, R. S., & Barto, A. G. (2014). *Introduction to reinforcement learning* (2nd ed.). MIT Press.

Świechowski, M., Godlewski, K., Sawicki, B., & Mańdziuk, J. (2023). Monte carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review*, *56*(3), 2497–2562. https://doi.org/10.1007/s10462-022-10228-y

The New York Times. (2022). *Wordle*. Retrieved June 7, 2025, from https://www.nytimes.com/games/wordle

Towers, M., Kwiatkowski, A., Terry, J., Balis, J. U., De Cola, G., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J. J., Tan, A. J. S., & Younis, O. G. (2024). Gymnasium: A standard interface for Reinforcement Learning environments. *arXiv*. https://doi.org/10.48550/arXiv.2407.17032

United Nations. (2015). *Transforming our world: The 2030 agenda for sustainable development*. Retrieved June 13, 2025, from https://digitallibrary.un.org/record/3923923?v=pdf

APPENDIX

A

DISTRIBUTION OF SHAPES IN THE DAILY BOARDS

Throughout this thesis, we use daily boards to test the performance of different models. Since machine learning models require large amounts of training data and because daily boards are reserved for testing, we must obtain training data elsewhere. We hypothesize that we can generate data for ourselves by sampling from a 2-dimensional discrete uniform distribution. To confirm this hypothesis, we check that the daily boards are generated in the same way. This involves hypothesis tests on two important assumptions: that each individual shape is generated from a uniform distribution (Appendix A.1) and that there is no correlation between neighboring shapes on the grid (Appendix A.2).

A.1 Hypothesis Test on the Uniform Assumption

Let X_k denote the k-th shape, $k = 1, \ldots, K$, with $K = 9 \cdot 7 \cdot n_{\text{boards}}$. Here, $n_{\text{boards}} = 100$ is the number of daily boards that we have used for this purpose. We perform the hypothesis test

$$H_0 : X_k \sim \text{DU}(0, 3) \quad \text{versus} \quad H_1 : X_k \not\sim \text{DU}(0, 3),$$

where $\text{DU}(0, 3)$ is the discrete uniform distribution with state space $\{0, 1, 2, 3\}$. Let N_j denote the total number of shapes of type j observed,

$$N_j = \sum_{k=1}^{K} \mathbb{I}(X_k = j),$$

where $\mathbb{I}(X_k = j)$ is the indicator function. Under H_0, we expect to observe equally many of each shape, that is,

$$\mu = E[N_j \mid H_0] = \frac{K}{4}, \quad j = 0, 1, 2, 3.$$

Thus, the test statistic

$$V = \sum_{j=0}^{3} \frac{(N_j - \mu)^2}{\mu} \sim \chi^2_3 \mid H_0.$$

From the 100 daily boards, we observe $N_0 = 1562$ pink shapes, $N_1 = 1606$ blue shapes, $N_2 = 1561$ green shapes, and $N_3 = 1571$ orange shapes, which gives $V = 0.85$. The corresponding p-value is then

$$p = P(V > 0.85 \mid H_0) \approx 0.84,$$

indicating that we do not reject H_0 under any reasonable significance level. Based on this result, it is reasonable to assume that each shape in the daily board is drawn from a discrete uniform distribution.

A.2 Hypothesis Test on the Noncorrelation Assumption

To check for spatial correlation, we use a two-step procedure. The purpose of this procedure is to check if a shape is correlated to its *neighbours*, which are the (up to) four adjacently connected shapes. First, we calculate p-values for each board individually, based on permutation tests with Moran's I as the test statistic (Cliff & Ord, 1981). Thereafter, we use Fisher's combined probability test to combine the 100 p-values into one, which indicates whether the shapes on the daily boards are correlated (Fisher, 1970).

For a board with 63 shapes, the observed Moran's I is defined as

$$I_{\text{obs}} = \frac{63}{W} \cdot \frac{\sum_{i=1}^{63} \sum_{j=1}^{63} w_{ij}(x_i - \bar{x})(x_j - \bar{x})}{\sum_{i=1}^{63}(x_i - \bar{x})^2}.$$

Here, $x_i \in \{0, 1, 2, 3\}$ is the shape at index i of the flattened board, $\bar{x}$ is the average shape, w_{ij} is 1 or 0 depending on whether x_i and x_j are neighbors or not, and W is the sum of all w_{ij}. The expected value of Moran's I when there is no spatial correlation is $E[I] = -\frac{1}{63-1} = -\frac{1}{62}$. By using Moran's I as test statistic, we can formulate the test of non-correlated shapes as

$$H_0 : I = -\frac{1}{62} \quad \text{versus} \quad H_1 : I > -\frac{1}{62}.$$

We approximate the p-value of this test for each board j, $j = 1, \ldots, n_{\text{boards}}$. To do so, we first calculate the observed statistic $I_{\text{obs},j}$ based on board j, and then calculate $I_{r,j}$ for $r = 1, \ldots, R$ based on $R = 10\,000$ permutations of the same board. Then, an estimated p-value for the test on board j is

$$p_j = \frac{\sum_{r=1}^{R} \mathbb{I}(I_{r,j} \geq I_{\text{obs},j})}{R}.$$

After calculating all 100 p-values, we combine them to one using the Fisher combined probability test statistic (Fisher, 1970),

$$U = -\sum_{j=1}^{n_{\text{boards}}} \ln p_j \sim \chi^2_{2n_{\text{boards}}} \mid H_0.$$

For the 100 daily boards with $R = 10\,000$ permutations each, we obtain $U = 206.43$. With $U \sim \chi^2_{200}$ under H_0, this gives a final p-value

$$p_{\text{tot}} = P(U > 206.43 \mid H_0) \approx 0.36.$$

Thus, it is reasonable to assume that the neighboring shapes on the daily boards are not correlated.

Since both the assumption on uniformly sampled shapes and the assumption on non-correlated shapes within a board hold, we conclude that it is likely that the daily boards published by NRK are sampled from a discrete uniform distribution.

APPENDIX

B

GITHUB REPOSITORY

This appendix includes the link to the GitHub repository in Appendix B.1, where all code and some relevant data are included. More information about this in the README file. Additionally, in Appendix B.2, we show some example images with explanation of the graphical user interface (GUI) for *Former* that we have implemented, where the player can choose any of the 100 daily *Former* games we have used for analysis in this thesis, or input a custom board. Moreover, the player can select a time limit for MCTS, and get recommended actions from the best-performing solver that we have created. For any of this to work, the C++ implementation of the game must be compiled. Further instructions for this is also in the README-file.

B.1 GitHub Repository Link

Code can be accessed from the link below, where a README file provides further instructions.

- https://github.com/espenurheim/FormerMSc.git

Alternatively, to clone the repository directly from the terminal, navigate to the target directory, and run:

```
git clone https://github.com/espenurheim/FormerMSc.git
cd FormerMSc
```

B.2 Play *Former* with Solver Recommendations

Here we display some screenshots of the *Former* GUI with explanations. When the `PLAY_FORMER.py` file is ran, an interface pops up, where the player can scroll and select among any of the 100 daily boards, or select a custom board (Figure B.1).

After selecting a board, the player is faced with the game interface (Figure B.2), which includes a board where each group has a corresponding number, and room for some information on the right. Initially, this information is the date, the best-known solution to the corresponding board, and the number of moves used so far. From here, the player can click on any shape on the board to remove the corresponding group, which immediately changes the board according to the game physics. Or, the player can type

Figure B.1: The initial board-selection display. The player can scroll to select any of the 100 daily boards used for analysis in this thesis, or select a custom board. The date and the best-known solution to each board is shown.

any time limit (in seconds) in the box at the bottom of the information page (where we in Figure B.2 have typed '1'), and perform a search with MCTS plus the $f_{\pi,(64,10)}$ solver with the given time limit. The game can also be reset to its initial board at any time.

After entering a time limit and pressing "Search", MCTS searches for the given time limit, before it outputs a recommended action and the maximum number of remaining moves if its recommendation is followed. This is displayed in Figure B.3. From here, the player can select an action or perform a new search with any time limit if the number of moves remaining is not satisfactory. If the recommended move is chosen, the recommended group is immediately updated to the next action in the sequences of actions found by MCTS, as shown in Figure B.4. If the recommended action is not chosen, the recommended tab disappears, and the player must start a new search to obtain a recommendation.

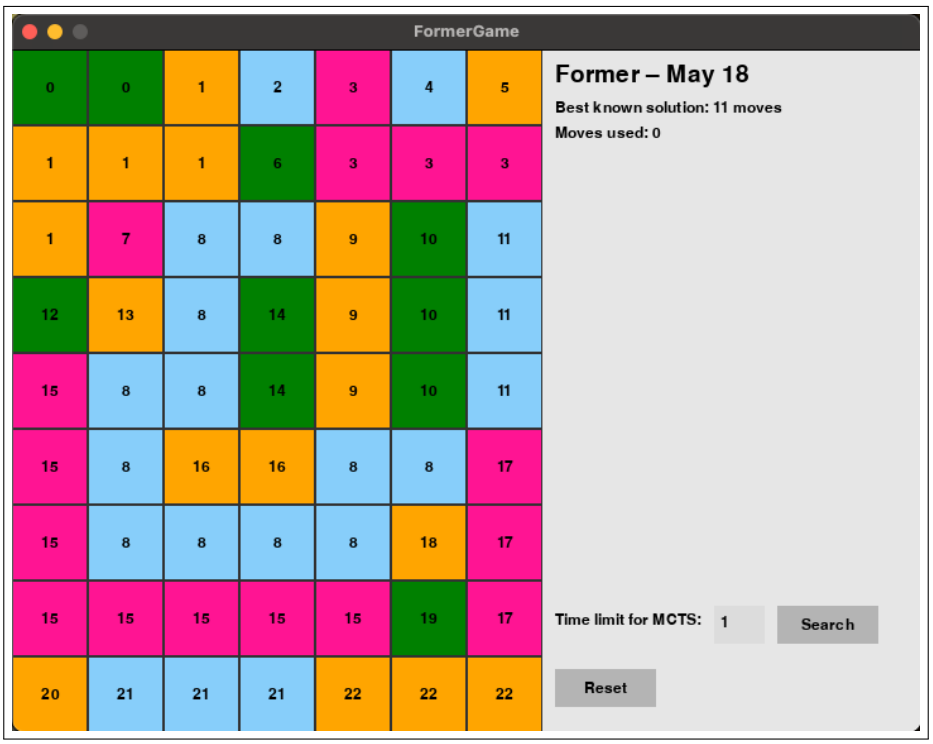


Figure B.2: The GUI after selecting the May 18$^{\text{th}}$ board in the initial board-selection display. The colored part of the screen is the selected board, where each group of shapes has been assigned its own number. On the right, information on what date it is, the best-known solution for the current board, and the number of moves used, are shown. Additionally, the player can run a MCTS solver with any time limit (in seconds), and obtain a recommended choice of action.

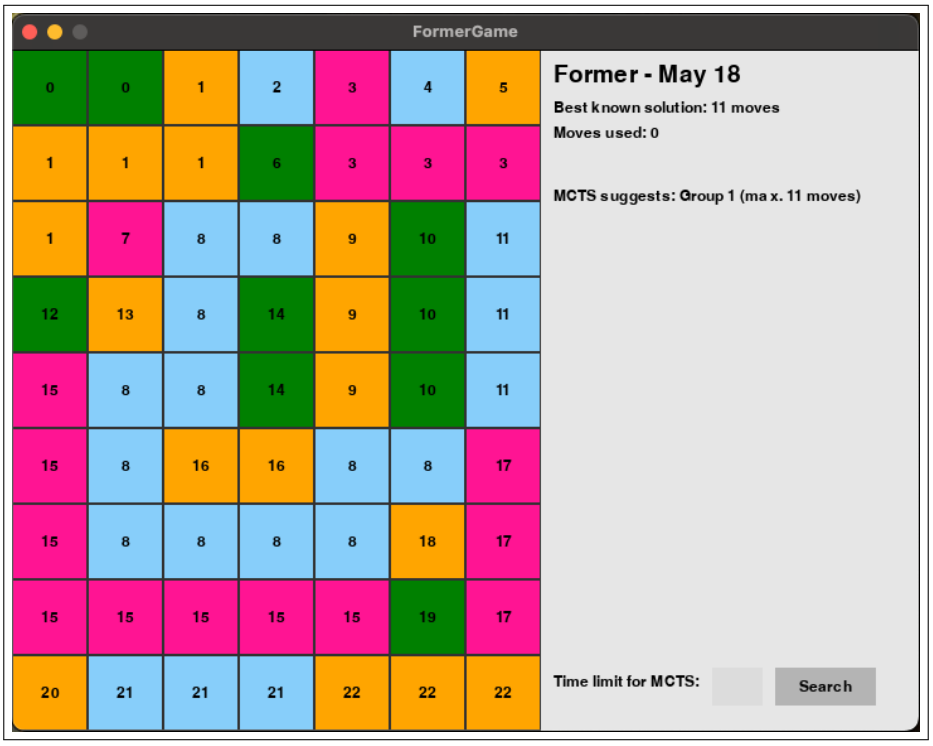


Figure B.3: The GUI for the May 18th *Former*-board after running MCTS for 1 second. MCTS suggests a group to be removed, and displays the maximum number of moves remaining from the current state if that move is selected first.

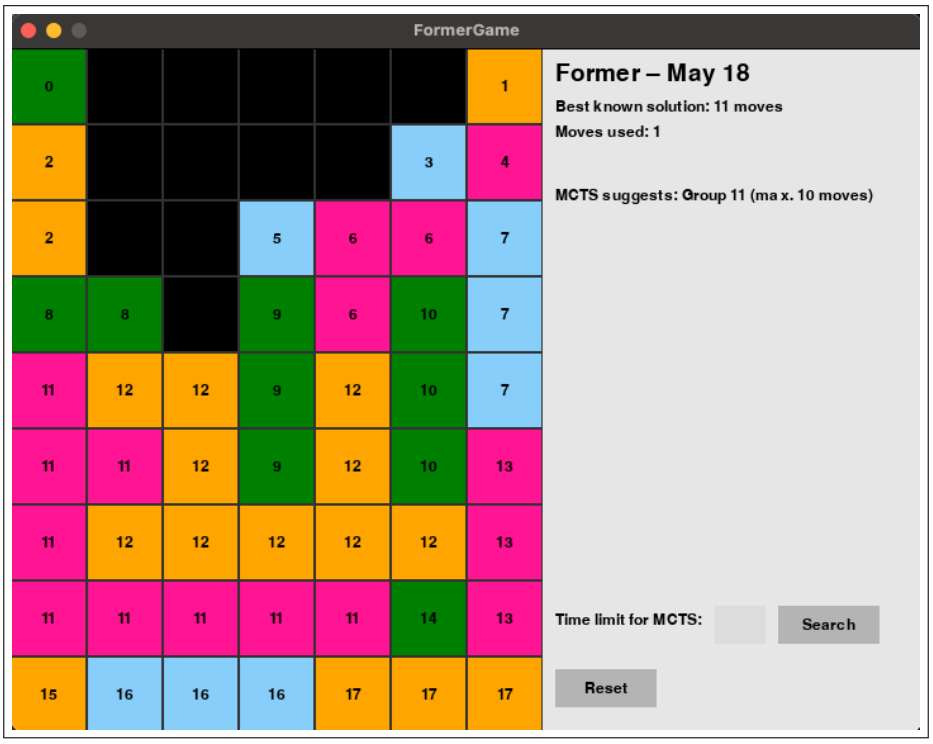
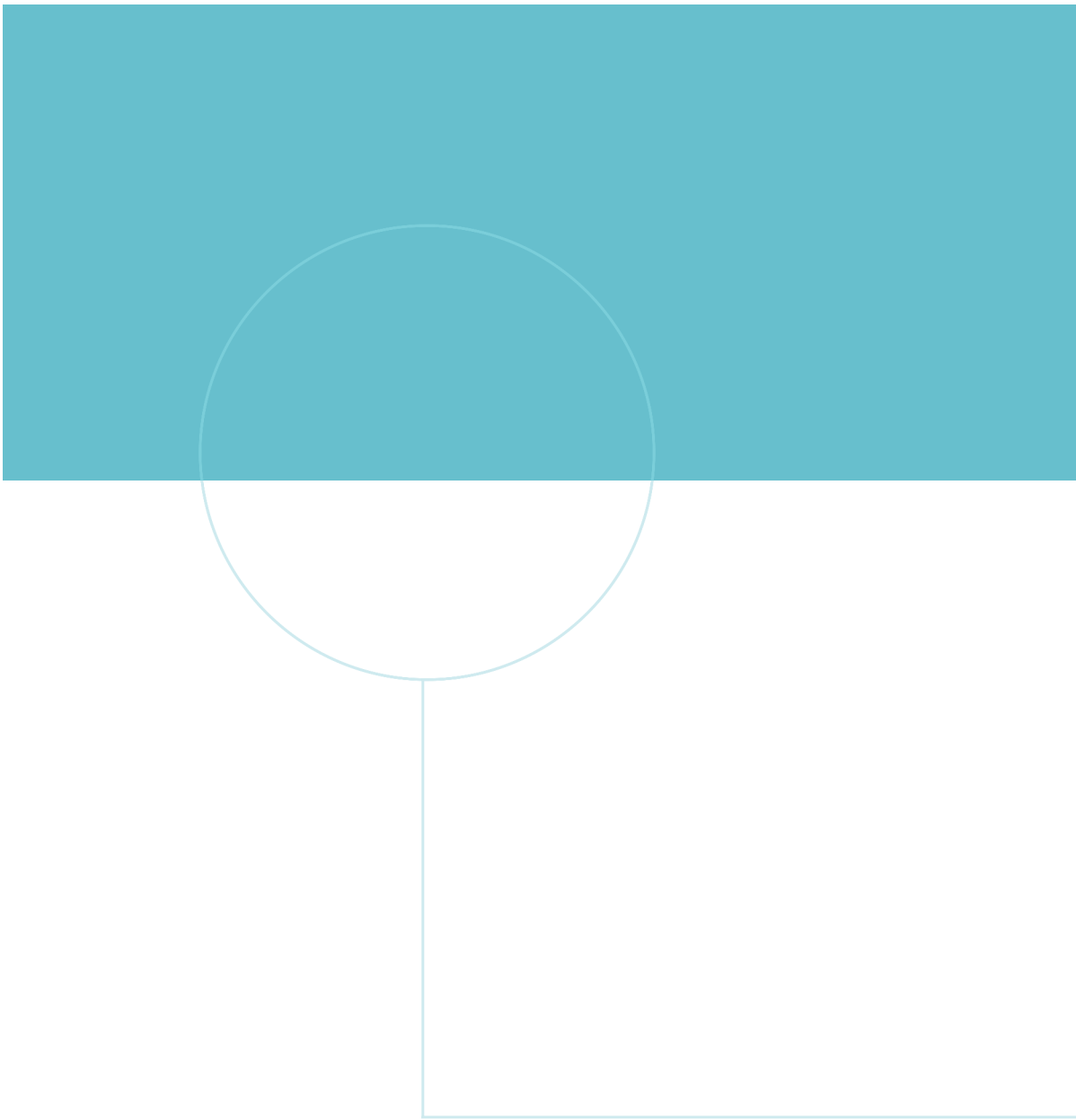


Figure B.4: The GUI for the May 18th *Former*-board after choosing the initial action recommended by MCTS in Figure B.3. The suggestion is updated, along with the maximum number of remaining moves according to the solver.



NTNU

Norwegian University of
Science and Technology