Maximilian A H Clark

Mastering Sevn using Reinforcement Learning

Computer Science Tripos – Part II

Pembroke College

2021

Declaration

I, Max Clark of Pembroke College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed *Maximilian Anthony Hugh Clark*

Date *31st July 2021*

Proforma

Candidate Number:	**2329F**
Project Title:	**Mastering Sevn using Reinforcement Learning**
Examination:	**Computer Science Tripos – Part II, July 2021**
Word Count:	**11478**[1]
Final Line Count:	**4626**[2]
Project Originator:	The dissertation author
Supervisors:	Prof. Neil Lawrence and Pierre Thodoroff

Original Aims of the Project

This project aims to implement DeepMind's AlphaZero to play the board game Sevn and investigate the feasibility of training AlphaZero without the use of elaborate hardware such as that used in DeepMind's implementation. This project also aims to assess the practicality of implementing AlphaZero with a graph convolutional neural network architecture (as opposed to the residual convolution network used by DeepMind) and discover the level of mastery achievable with these techniques and ordinary hardware in the game of Sevn.

Work Completed

The project was a resounding success. The AlphaZero algorithm, graph convolutional neural network, the game logic, user interface and many tools used for evaluation were all implemented successfully. With all of this completed, I was able to train the AI and assess the feasibility of doing so with publicly available hardware. All success and extension criteria were met and the AI reached an impressive standard of performance, compared to other agents and human play.

[1]This word count was computed using TeXcount Web Service (`https://app.uio.no/ifi/texcount/online.php`).

[2]This line count was computed using cloc (`https://github.com/AlDanial/cloc`)

Special Difficulties

None.

Contents

List of Figures

Acknowledgements

I owe great thanks to the following for their time and efforts in aiding my project and dissertation:

- **Prof. Neil Lawrence** and **Pierre Thodoroff** for their considerable dedication in supervising my project, which involved providing very valuable guidance for the implementation, teaching me technical writing and reviewing this write-up.
- **Anonymised group of students** for playing games against my AI to help me evaluate the project.

Chapter 1

Introduction

Computer scientists as early as Turing have been captivated by writing algorithms to play games. In 1948, Alan Turing and David Champernowne developed a set of rules to make moves in chess, called *Turochamp* [9]. Whilst the rules were too complicated to be run on the hardware of the time, Turing was unperturbed, given his confidence in the future progress of technology. However, for much of history, AI for chess proved to be a chimera for both the hardware and algorithms. It wasn't until 1997 when an artificial intelligence, Deep Blue [11], became the world chess champion having started development in 1985. The difficulty of creating an AI for chess is due to the immense number of reachable states in the game. This number, known as the **state-space size**, is estimated to be 10^{43} for chess [24]. Since Deep Blue, chess programs have long surpassed human ability. But it wasn't until recently when the same could be said about the game of go, with an incomprehensible state-space size of 10^{170} [28]. Given this complexity, hard-coded rules and heuristics no longer sufficed and mastering go required new innovations in generalised learning and planning.

In March 2016, with a score of 4 to 1, DeepMind's AlphaGo AI [25] beat reigning world go champion Lee Sedol [4]. Given the amount that professional go players rely on intuition, this was a historic milestone for artificial intelligence. In October 2017, DeepMind published their work on AlphaGo Zero [27], which, unlike AlphaGo, learnt to play *tabula rasa* (with no data or guidance from human play) and beat AlphaGo 100 games to 0. In December that same year, DeepMind published their work on AlphaZero [26], which involved tweaking AlphaGo Zero to generalise to any two-player, turn-based, perfect-information game. The success of AlphaZero was demonstrated by achieving a level of mastery in the games of chess and shogi (Japanese chess) never before seen in human or artificial intelligence play and matching AlphaGo Zero's performance in go.

The project this dissertation describes is an application of AlphaZero to the game Sevn (a variation of Paletto). Sevn is a two-player game about taking coloured tiles – see section 2.1 for an explanation of the game.

This project aims to make three main contributions to the field:

1. An application of AlphaZero to Sevn, being the first open-source AI for the game the dissertation author is aware of.
2. A combination of graph neural networks with AlphaZero to see what challenges and benefits arise.
3. An investigation of the feasibility of training AlphaZero with ordinary hardware.

1.1 Motivation

Games are useful as a testbed for AlphaZero since they are a contained environment where decision and planning principles can be tried and tested before being generalised to real-world problems. AlphaZero has indeed seen applications outside of games [13].

Go was an appropriate choice for DeepMind because "go is known as the most challenging classical game for artificial intelligence because of its complexity" [2]. It was previously unconquered territory and the perfect challenge to demonstrate the capability of AlphaZero given sufficient compute resources.

Sevn is an appropriate choice for this project for a number of reasons. With an average game length of 30 plies[1] and branching factor[2] of about 10, the state-space size is around 10^{30}. Compared to chess at 10^{43} [24], we expect training an AI to play Sevn to be more feasible, given the resources available, yet the problem is still intractable for a brute-force approach[3].

Secondly, Sevn provides an opportunity to utilise the AlphaZero algorithm with a novel neural network architecture: relational graph convolutional networks [22] (as opposed to the residual convolutional network used by DeepMind's implementation). Graph neural networks are an interesting application for this project because strategy for Sevn is rotationally and translationally invariant (see section 3.2) which can be exploited by a graph representation but not by a matrix. This project aims to assess the feasibility and benefits of combining the AlphaZero algorithm with graph neural networks.

See section 2.4 for further reasons for choosing Sevn.

While AlphaZero was undoubtedly very successful, the resources used for training[4] were far beyond what's available to most. This project aims to investigate the success of the AlphaZero algorithm given more limited resources. See the resources declaration in appendix B.

Finally, this project proposes creating the first open-source artificial intelligence for Sevn and questions whether human-level mastery of Sevn is achievable. Performance will be

[1] A ply is one move for one player.

[2] The branching factor is the average number of available moves at any one state.

[3] Modern hardware has been shown to search 20 billion chess states per second [7]; at that rate, exploring the entire tree of Sevn would take about 10^{12} years – 70 times the age of the universe.

[4] 5000 1st-gen TPUs for self-play and 54 2nd-gen TPUs for network training [26].

compared to a current general method for AI – the upper confidence trees (UCT) algorithm [17, 16]. UCT is a minimax approximator with a scalable amount of tree search, allowing one to balance performance with time/compute cost (section 2.2 describes the algorithm). AlphaZero and UCT are fundamentally similar algorithms which both make use of Monte-Carlo tree search (MCTS); their primary difference is AlphaZero's use of a function approximator to gain prior knowledge about the value of states and moves. UCT is strong when the game state tree is small as it can effectively identify sensible areas of the tree to search quickly, but can struggle when the the state space is too large. This is seen when playing Sevn, where questionable decisions are made early on. See section 4.1.1 for an example of this. This demonstrates the need for a function approximator to indicate the best areas of the tree to search early on. Therefore, we expect AlphaZero to outperform UCT, and if our results demonstrate this is achieved, it would be proof of a successful implementation and would be evidence for the feasibility of applying graph neural networks to AlphaZero with limited resources.

Chapter 2

Preparation

Before the principal work had begun, this project started with understanding the game of Sevn, the AlphaZero algorithm and the suitability of the algorithm for this particular game, as well as preparing software engineering tools and practices for the implementation.

2.1 The Game of Sevn

Whilst knowledge of or competence at the game is in no form required for the reader, an understanding of the game's rules is recommended for fully comprehending the motivation behind some decisions made in the implementation of the project. See appendix A for a full explanation of the game. Below is an overview.

Sevn is a two-player, turn-based strategy game about taking tiles. It involves a 7x7 grid of tiles, with 7 colours and 7 tiles of each colour. On their turn, a player can take corner tiles of the same colour. A player wins if they take all 7 tiles of a colour or they are the dominant owner of more colours than the opponent when all tiles are gone.

Figure 2.1, shows the board of an example starting state.

This game is flexible enough to played at different grid sizes – a fact that this project makes use of. This dissertation shall use the term *base number*, referring the size of the

Figure 2.1: The board of an example starting state. The first player may take the black (□) in the bottom left or any number of the three pink (△) tiles exposed as corners.

grid and the number of colours.

Sevn is a variation of the game paletto designed by Dieter Stein and exists as a mobile app Sevn on the iOS App Store. This app contains the only AI for Sevn which the author is currently aware of and is used as a benchmark to evaluate the AI in this project in section 4.1.2.

Before explaining the algorithms, it is useful to confirm some Sevn-specific definitions:

- *State*: the position and colours of the tiles (the board) and the number of each colour the players have taken (the score).
- *Terminating state*: a state which results in one player winning. *Terminating states* have no *actions*.
- *Actions*: from any state, the set of actions contains all possible ways of legally taking tiles from that state.

2.2 Upper Confidence Trees (UCT) Algorithm

The UCT algorithm [17, 16] operates in two stages: simulation and final move selection. The simulation stage is made up of Monte-Carlo tree search (MCTS) *samples* of the game state tree and, for each state s and action a encountered, records information about what is observed:

- $N(s, a)$ = number of times a has been played from s during simulation
- $Q(s, a)$ = mean evaluation of all samples which took a from s

Each sample terminates at a *terminating state* and generates a sample evaluation z which depends on the winner at the terminating state. The algorithm backtracks along the path the sample took, updating N and Q for each state encountered. The value z is used to update Q and is inverted each move such that $z = 1$ when the next player at state s was the sample winner and $z = -1$ otherwise.

During simulation, moves are selected using equation (1):

$$\underset{a}{\operatorname{argmax}} \left(Q(s, a) + C\sqrt{\frac{\sum_b N(s, b)}{1 + N(s, a)}} \right) \tag{1}$$

where C is a constant controlling the level of exploration and $\sum_b$ is shorthand for summing over all actions available from state s. The intuition of this formula describes a balance between exploitation of nodes which have shown promising results ($Q(s, a)$) and exploration of nodes with low visit counts ($C\sqrt{\ldots}$).

Simulation continues until states have been visited a predefined number of times. After which, equation (1) is used for the final move selection.

2.3 AlphaZero Algorithm

Reinforcement learning (RL) has the potential to change how AI is written for games. AlphaGo Zero demonstrated that supervised learning (which has been the traditional method for such tasks) is not a viable route to true mastery of a complex game; if it's possible to build a knowledge base of sufficient quality, it would require far more effort and result in far less generalisation than an RL approach. Supervised learning is appropriate for recognizing cat images since people can perfectly perform the task and thus building a good knowledge base is achievable. However, the recent performance of AI in games like chess and go demonstrate that perfect mastery of games is not achievable for humans. This suggests we ought not let human play influence the AI but instead use reinforcement learning where the AI discovers its own strategy driven by a reward.

Instead of a knowledge base, self-play is used to generate training data. Training via self-play is important because it is guaranteed to be a challenging match and there is no risk of memorising the strengths/weaknesses of a particular opponent; for any strengths the AI discovers, it is forced to find a strategy to defend against it. The training data is used to train a neural network which estimates the value of states and actions.

What if we used the neural network's outputs to decide moves? Then the quality of its play will only be as good as the quality of the data used to train it. This means the self-play-train cycle won't result in improvement. AlphaZero solves this by using the network to guide exploration of the game state tree which is then capable of play stronger than the raw network. So, the network learns from the tree search, which is always better than the network. Therefore, in theory, if the network can continually learn to approximate the improved play of the tree search, an optimal agent will be achieved.

The AlphaZero algorithm is described in full in DeepMind's AlphaGo Zero paper [27]. Note that the algorithm used in AlphaGo Zero is the same as AlphaZero with minor changes to the training process [26] for generalisation.

Like UCT, the algorithm uses MCTS to explore the game state tree. Unlike UCT, a function approximator f_θ with parameters θ is used to guide the tree search.

For game state s, let $\alpha_\theta(s)$ be a function which returns the search probabilities – a vector $\boldsymbol{\pi} \in \mathbb{R}^n$ where n is the number of possible actions (moves) from state s.

The function approximator outputs two values: $f_\theta(s) = (\boldsymbol{p}, v)$, where move probabilities $\boldsymbol{p} \in \mathbb{R}^n$ is a prediction of $\boldsymbol{\pi}$ and $v \in \mathbb{R}$ in the range $[-1, 1]$ is the predicted winner of game given the current state inputted. Output $\boldsymbol{p}$ is referred to as the *policy* and v is referred to as the *value*.

The AlphaZero algorithm [26] has many similarities to the UCT algorithm. Simulation computes $\alpha_\theta(s)$, during which it stores for each state s and action a encountered:

- $N(s, a)$ = number of times a has been played from s during simulation
- $Q(s, a)$ = mean evaluation of all samples which took a from s
- $P(s, a)$ = the value $\boldsymbol{p}_a \in \boldsymbol{p}$ generated by $f_\theta(s)$

Unlike UCT, a sample may terminate before a *terminating state* is reached. If a sample encounters an unseen state s, the sample terminates and evaluates the new state: $f_\theta(s) = (\boldsymbol{p}, v)$. It assigns $P(s, a) = \boldsymbol{p}_a$ for each action a from s and generates the sample evaluation $z = v$. If a terminating state is reached then z is assigned in the same way as the UCT algorithm. In either case, z is inverted like in the UCT algorithm and backtracking is used to update the Ns and Qs.

During simulation, moves are selected using equation (2):

$$\underset{a}{\operatorname{argmax}} \left(Q(s,a) + c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \right) \tag{2}$$

where c_{puct} is a constant controlling the level of exploration. Like UCT, this formula balances exploitation and exploration.

Simulation continues until predefined number of samples have been performed[1]. The simulation returns the search probabilities using equation (3):

$$\boldsymbol{\pi} = \left[\frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}} \text{ for each action } a \text{ from } s \right] \tag{3}$$

where τ is a parameter to control exploration.

If AlphaZero is playing competitively, the move chosen is $\underset{a}{\operatorname{argmax}}\, \boldsymbol{\pi}_a \in \boldsymbol{\pi}$. The value of τ is irrelevant in this case.

If AlphaZero is training, a move is randomly chosen with probabilities proportional to those in $\boldsymbol{\pi}$. This is to encourage exploration so the algorithm is trained for all areas of the tree.

AlphaZero's tree search is effective when the game state tree is large because it makes use of f_θ to identify sensible areas of the tree to search. The *policy* reduces the breadth of the search and the *value* reduces the depth of the search.

2.4 Appropriateness of a AlphaZero-style Solution

Deep learning is shown to be the best current approach for games such as chess and go [26]. Several aspects of Sevn are complementary for writing an AlphaZero-style AI:

- Sevn is a classical game. In other words, Sevn is two player and turn-based with perfect information and a perfect simulation.
- The number of moves in a game is naturally capped at 49 ply by the number of tiles, though is more typically around 30.
- The game state tree is acyclic.

[1] DeepMind's implementation used 1600 samples [27, 26]

- There is no draw-state – one player must win.
- The complexity of the game can be scaled by selecting a base number (which must be odd to retain the previous property). This means we can aim for a general AI capable of playing games of different sizes and difficulty – section 3.2 describes how this is accomplished.

2.5 Requirements Analysis

The success and extension criteria mentioned in the original project proposal (appendix B) revolve around creating an AI which meets particular milestones of performance. In order to meet these criteria, the implementation must include both the tools to play/train the AI as well as tools to evaluate it. These can be further split into more specific deliverables, specified in table 2.1.

Table 2.1: Project deliverables

Deliverable	Priority
AI Implementation	
Implementation of the game logic	Fundamental
Appropriate representation of the game state	Fundamental
A neural network accepting the game state representation as input	Fundamental
The AlphaZero algorithm	Fundamental
The training loop: initiating self-play, generating training data and training the neural network	Fundamental
Evaluation	
The UCT algorithm	Fundamental
A system to battle agents against each other	Fundamental
An automated system to battle agents against each other	Preferable
A ranking/rating system to compare agents which haven't played each other	Preferable
A user interface to allow the AI to play humans	Preferable
iOS screen parser to battle the AI from the Sevn app	Preferable
Code to measure and compare agents on move times	Preferable

Requirements Justification

Given that the project is investigating the AlphaZero algorithm specifically, there are no alternatives to the deliverables mentioned under *AI Implementation*.

The UCT algorithm was chosen due its similarity to the AlphaZero algorithm; it is a general approach which does not require human expertise in the game. AlphaZero surpassing

UCT is a demonstration of successful implementation and shows feasibility of AlphaZero in the selected conditions, as explained in section 1.1. A Stockfish [1]-style implementation was not chosen because it would involve meticulously hand-crafting heuristics specific to Sevn (Stockfish involved a huge collaboration between chess grandmasters and computer scientists) and the dissertation author does not pretend to have the competency in Sevn necessary for such a task.

The Elo rating system [8] was selected to rate and compare agents. Alternatives to the Elo rating system exist, such as TrueSkill [15] and its extensions [20], but the Elo system was chosen due to it being a well tried and tested method for board games such as chess (the word Elo is often associated with chess [3]) and the main evaluation system used in DeepMind's AlphaGo Zero report [27].

The AI from the Sevn app was chosen for evaluation because, despite not knowing anything about how it works, it is the only existing AI specifically crafted for Sevn which the dissertation author is aware of at the time of writing.

2.6 Software Engineering Tools and Techniques

Language and Modules

Python was the chosen programming language. Python has arguably the most support for machine learning projects with a variety of libraries which are very useful for this kind of project. Crucially, the Python module `DGL` [29] (deep graph library) has support for creating relational graph convolutional networks, which is not yet a standard in all ML libraries. `DGL` works with both PyTorch [21] and TensorFlow [6]. PyTorch was chosen because you can manipulate the computation graph on-the-go, as opposed to defining the computation graph from the start, as done in TensorFlow. Secondly, PyTorch is said to be more Pythonic [5].

Jupyter notebooks were used to run the training and primary evaluation because they are the interface for connecting to Google Colab instances. Google Colab was chosen because it offers publicly available hardware, allowing us to investigate the feasibility of AlphaZero using ordinary compute power. Secondly, Colab helps make the project accessible and reproducible. Section 4.5 assesses the success of this reproducibility.

The repository itself was written using Visual Studio Code because of familiarity and a smooth integration with developer tools like *Git*, which was used in this project for version control.

Version Control and Backup

A remote repository was created on GitHub where around 150 commits were pushed. A further copy of the repository was kept on Google Drive. As well as acting as a second backup for the code, it allowed me to connect Jupyter notebooks on Google Colab to the repository.

Repository Management

To organise the project and manage dependencies, I used Poetry. This tool automatically created a virtual Python environment with all dependencies installed and allowed me to specify many entry points into the repository so I could run various tasks from the command-line like formatting, training, playing or plotting various graphs. I used the Python module `black` to auto-format the repository.

I used docstrings to describe functions and classes to make the code more readable.

The entire repository was written with extensibility in mind e.g. the game logic allowed for any combination of agents to play each other and the AlphaZero implementation allowed for any function to act as the *prediction* function approximator.

2.7 Starting Point

This project does not build on any other code/project, but only makes use of common libraries such as DGL, PyTorch, Numpy, Pygame (for the UI) and Matplotlib (for plotting the graphs in this dissertation).

2.8 Summary

This project requires an understanding of the rules of Sevn and the AlphaZero/UCT algorithms before implementation can begin. A professional approach to developing the project is imperative to ensure that the complex algorithms are implemented cleanly and correctly and the subsystems interact as expected.

With these concepts understood, work can begin on the implementation to work towards the goals laid out in the motivation section 1.1.

Chapter 3

Implementation

This chapter explains what was produced during this project. Whilst every deliverable described in table 2.1 was completed fully, efforts were primarily focused on the design of the graph neural network architecture, its input format, the correctness of the AlphaZero algorithm and the tools to evaluate the agents produced.

The core component of the implementation is the architecture of the AI and how it trains.

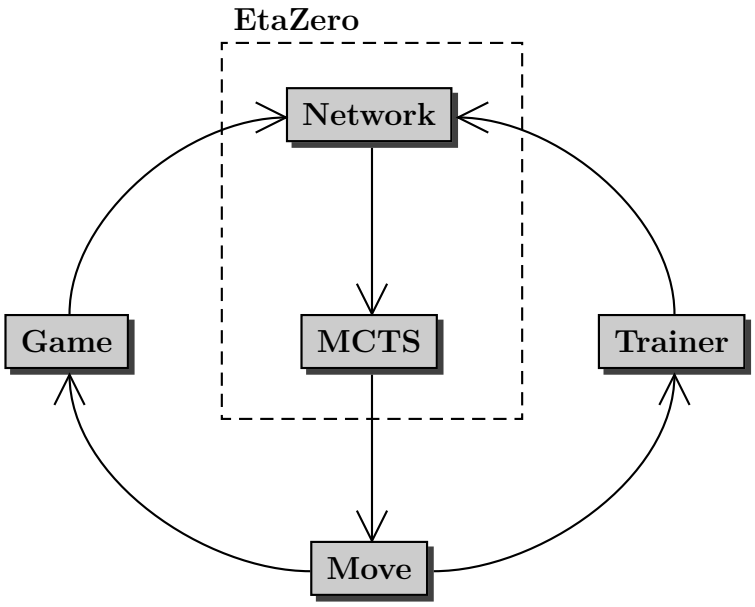

Figure 3.1: Core information flow overview. Arrows show the direciton of information flow. Note this is a simplification.

Figure 3.1 shows how EtaZero (described in section 3.4.1) is split into two components. In the repository, the MCTS (AlphaZero algorithm described in section 2.3) is implemented in the class `EtaZero` as described in 3.4.1. This class is composed with an instance of the `Network` abstract class. The implementation used was class `PolicyValRGCN` – described in section 3.3.4. The output of the network is used to guide the MCTS which ultimately selects a move.

During both training and competitive play, the selected move is passed to the `Game` implementation (described in 3.6). This results in a new game state. As the MCTS explores the game state tree, game states are transformed to an appropriate representation as in section 3.2 and used as input to the network.

During training, a game plays out until completion, after which the training data is created from the moves made. Once enough games have been played, the network parameters are updated using the training data. The `Trainer` class responsible for this is described in section 3.3.5.

3.1 Repository Overview

The project was split into many files and organised into a directory tree allowing for cleaner commits and a project which is easier to understand. The overview is shown in table 3.1. All code in the repository was written by the dissertation author.

Repository Justification

The `src/` directory was used to separate the source code from the build system and allow extensibility in case a `tests/` directory was required[1]. Within `src/`, files were grouped by purpose and inheritance; for example, the agent abstract class and all of its implementations are kept in the `agents/` subdirectory.

An important separation is the between the reinforcement learning (in `networks` and `agents`) and the environment (in `game/`). This makes these two components each more modular, in case different approaches ever need to be applied to the environment or the same RL approach to a different environment.

3.2 Game State Graph Representation

Formatting the game state as a matrix may not be the most appropriate for our case. Encoding the game as a graph enables us to exploit various kind of invariance that a matrix representation cannot. The strategy of Sevn has translational, rotational and other kinds of invariance. For example, the boards in figure 3.2 are, strategically speaking, identical.

That's because two corner tiles are takeable regardless of their position and they do not reveal other tiles once taken. As such, we are only interested in how tiles conceal other tiles i.e. the tile relations, not the tile positions. Ideally, we want to choose a representation format[2] such that each strategically distinct state has exactly one representation e.g. all states in figure 3.2 should have the same representation. Formally, for the set of distinct

[1]Unit tests are useful for defining correct behaviour and hence beneficial for multiple people working on the project or for understanding past code. Unit tests were not created because code correctness was ensured through manual testing so directing efforts elsewhere resulted in a more significant project.

[2]Note the use of the word *representation* in this section refers to the input into the neural network (e.g. matrix or graph) and not the network's internal representation of the state.

Table 3.1: Repository overview.

Directory		Description
`scripts.py` and makefile[3]		Organises project entry points and automates formatting.
`src/`	`agents/`	Agent abstract class and implementations incl. EtaZero.
	`data/`	Stores training data, elo ratings, model checkpoints and timing data.
	`evaluation/`	Scripts to evaluate agents e.g. by performance, time.
	`game/`	Implementation of game, state and UI.
	`ios_screen_capture/`	Parses state of Sevn iOS app.
	`networks/`	Network abstract class, neural network implementations and trainer.
	`utils.py`	Commonly used functions e.g. loading model checkpoints.

representations X and the set of strategically distinct game states Y, there ideally exists an invertible bijective function $f : X \leftrightarrow Y$.

With Z as the set of policy-value outputs, let $g : Y \to Z$ be a hypothetical function which implements an optimal strategy. Since we feed our choice of representation as input, the domain of the neural network is X and it learns to approximate $g \circ f$.

If f is non-surjective (one-to-many), we have not specified enough information for the AI to make appropriate decisions. If f is non-injective (many-to-one), the neural network will likely produce different results for strategically identical positions – the domain of $g \circ f$ will be larger than it needs to be. This can, in some cases, be remedied using data augmentation (e.g. for the matrix case, duplicate the training data four times and apply four distinct rotations to account for rotational invariance). Augmentation becomes impractical when we look at other invariances. For example, each colour has the same value in Sevn. If they are one-hot-encoded in a matrix, we can make an augmentation for each permutation of the colours. For base 7, that's $7! = 5040$. Consider a state with one remaining tile and the score value for each colour unique. To account for translational and colour invariance, we must augment this one data example to $49 \times 7! = 246,960$ examples. This is why we aim for a representation format which encompasses the invariances.

[3]The project was organised using *Poetry* (`https://python-poetry.org/`). The makefile equivalent is `pyproject.toml`.

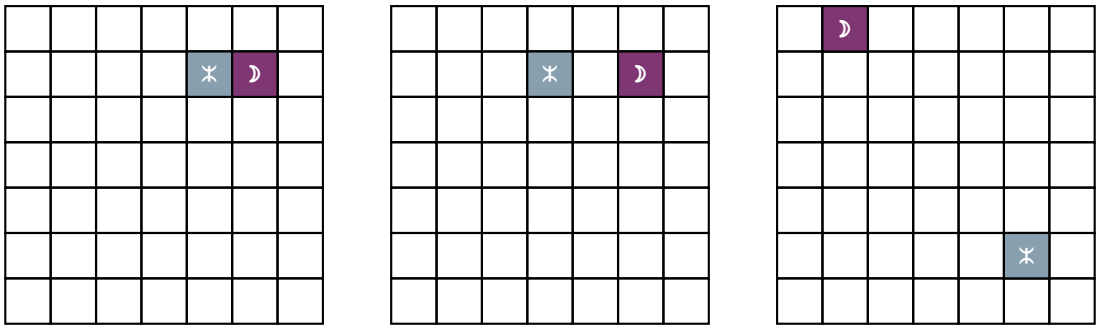

Figure 3.2: Examples of strategically identical boards in Sevn.

3.2.1 Tile Nodes and Relations

To encapsulate the idea of relational rather than positional tiles, a graph representation can be used [30]. A second reason for using a graph as input is that we can aim to design a neural network capable of processing game states of any base number, since unlike a standard convolutional neural network, the size of the input is not fixed.

We require a directed edge *hidden-by* which points from a concealed tile to a tile which could potentially reveal the tile if it were taken. Figure 3.3 shows some examples with their corresponding graph. Another relation *conceals* is used as the dual of *hidden-by* so that information can pass both ways in the neural network.

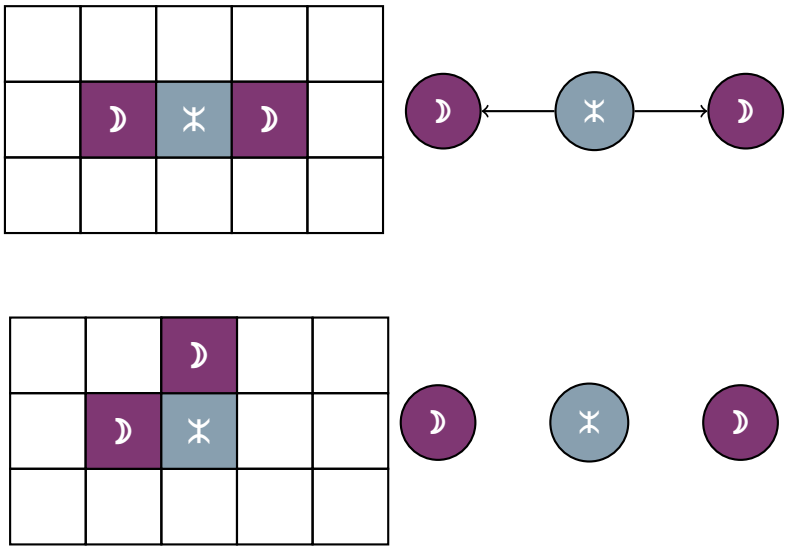

Figure 3.3: States and their graphs demonstrating *hidden-by* relations. The second example shows no relations since all of the tiles are exposed and takeable.

This alone still excludes positional information crucial for strategy. When a tile has all 4 sides covered, it will be revealed if we take 2 of the neighbouring pieces, so long as they aren't opposite each other. A graph won't tell us the orientation of these neighbours unless we specify a new edge, *diagonal* which is *only* used to connect the neighbours of a tile concealed by 4 tiles and is undirected. See figure 3.4 for examples.

The algorithm to generate a graph with positional relations is then:

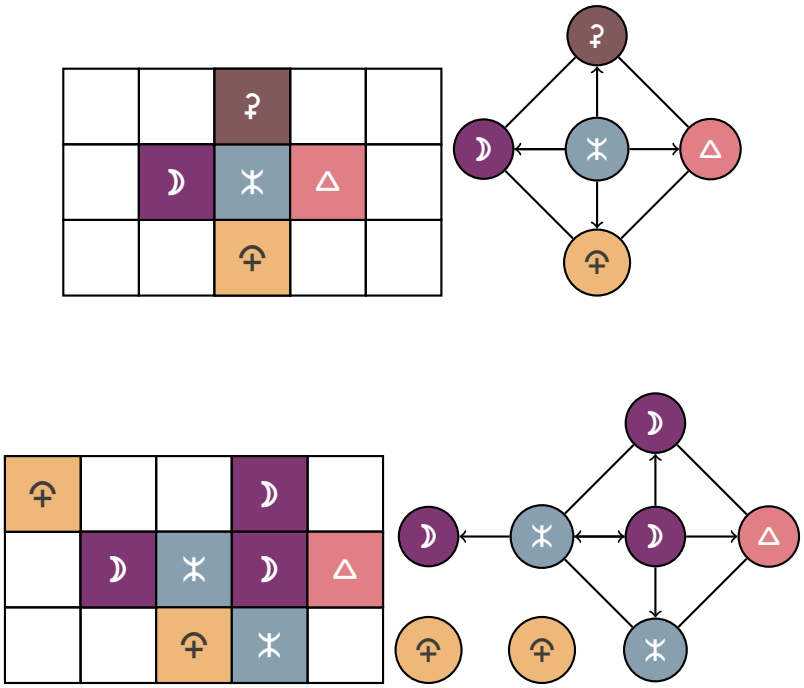

Figure 3.4: States and their graphs demonstrating *hidden-by* (directed edges) and *diagonal* (undirected edges) relations.

1. For every tile, add a node.
2. For each tile which is *not* takeable, add pairs of *hidden-by* edges to opposite neighbours *only*. E.g. if there are 3 neighbours, only add 2 edges since one of the neighbours is independent, strategically speaking.
3. For each *hidden-by* edge, add a *conceals* edge between the same nodes but in the other direction. For simplicity, *conceals* edges are not shown in the diagrams.
4. For every tile with all 4 sides covered, add *diagonal* edges connecting its 4 neighbours cyclically.

For information to propagate throughout the graph neural network, the graph must be fully connected. As such, we add a new edge *takeable* which densely connects all takeable nodes. This will result in a connected graph; it's clear to see that any isolated group of connected nodes must have takeable nodes. Secondly, this relation explicitly encodes which tiles are takeable, which is vital for strategy.

The AlphaZero algorithm requires a function approximator [26] which takes the game state as input and outputs both a value (estimated winner) and a policy (vector of move probabilities – one for each possible action). In order for the graph neural network to output a policy vector, we add an artificial node to the graph for every possible move from the current state. Let's call these nodes *action nodes*. We connect these nodes to the graph by adding an edge *from-action* from the node to each tile that would be taken by that move and an edge *to-action* going the other way.

Lastly, we need to encode the colour of the tiles into our representation. Whilst we

could one-hot-encode the colours into the nodes' feature vectors, this adds unnecessary information – f would be a surjection; no colour is inherently more valuable than another and switching any pair of colours and their corresponding scores bears no affect on the strategy. To get around this, we can instead add a new edge *same-colour* which, for each colour, densely connects all nodes of that colour.

The seven proposed relations are:

- *Hidden-by* (directed)
- *Conceals* (directed)
- *Diagonal* (undirected)
- *Takeable* (undirected)
- *Same-colour* (undirected)
- *To-action* (directed)
- *From-action* (directed)

3.2.2 Node Feature Vectors

Each node will contain a vector of values, completing the input for the graph neural network. We need to encode scoring information relevant to strategy.

Let n be the base number and p_i and q_i be the number of tiles of colour i taken by the current player and the current player's opponent respectively. In order to create a network general enough to play any board size, we must not rely on the network knowing the base number for strategy. The node vector for a tile of colour i will be:

[*player's colour score, opponent's colour score, score difference*]

- *Player's colour score:* $n - p_i + q_i$. For each remaining colour, a player is interested in how many tiles of that colour they need to get to own all of them. This equals the base number minus the number they have taken so far, but should increase if the opponent takes tiles of that colour. We can store this value as it's independent from the base number.
- *Opponent's colour score:* $n + p_i - q_i$. This is the same as above, but concerning the player's opponent.
- *Score difference:* We also need to encode the number of colours each player has a majority on (since all nodes of a colour can be removed, depriving the network of that information). For this, we can provide a score difference equaling the number of colours the current player has a majority on minus that of the opponent. This is not specific to any node but will be included with every node.

The colour scores are useful because they equal zero when a player has taken all tiles of a colour regardless of n. If the number of remaining tiles of a colour is not equal to a

player's colour score for that colour, then that means the opponent has taken tiles so it's no longer possible to take all tiles of that colour.

The final layer will output a vector of size 2 for each node which will encode the policy and value.

Such a neural network ought to be general enough for a Sevn game of any base number. The success of this is evaluated in section 4.3.

3.3 Neural Networks

There are various approaches to AI for games which don't involve neural networks. For example, Stockfish [1] is a chess engine which relies on many heuristics and alpha-beta pruning. A weaker but more general AI is the UCT agent as described in section 2.2.

These approaches are either (as in the case of Stockfish) too specialised, require a lot of knowledge from experts in the game and are not applicable to other games, or (as in the case of UCT) are weak when dealing with non-trivial cases; with limited compute resources, UCT struggles to correctly evaluate early-game positions due to the state-space size of the remaining tree and its inability to estimate the value of an unseen position.

Neural networks solve these problems by acting as a function approximator to learn game strategies (meaning no human expert knowledge is required) and thus can evaluate unseen states.

This section explains policy-value networks and value-win networks, which define the network outputs. Each of these can be 2D convolutional networks or graph neural networks, which are also described in this section.

3.3.1 Policy-Value Network

As explained in section 2.3, DeepMind's AlphaZero used a policy-value network to guide the MCTS. The network receives a game state as input and outputs a policy-value pair.

The biggest challenge with AI for games like chess and go is the enormous state-space size. Exploring the state tree is futile unless the search can be guided to sensible areas of the tree. The policy-value network enables search of the tree – the *policy* reduces the breadth of the tree and the *value* reduces the depth.

3.3.2 Value-Win Network

The downside of a neural network which returns a policy vector is that designing an architecture to return a vector of variable size is cumbersome. For example, the methods section of the AlphaZero paper [26] explains how the 4,672 possible chess moves are one-hot-encoded in the output vector. This project, using a graph neural network, has the ability to use nodes to represent values in the vector and so the size of the vector can be scaled. I created artificial *move nodes* to encode the policy, as explained in section 3.2.

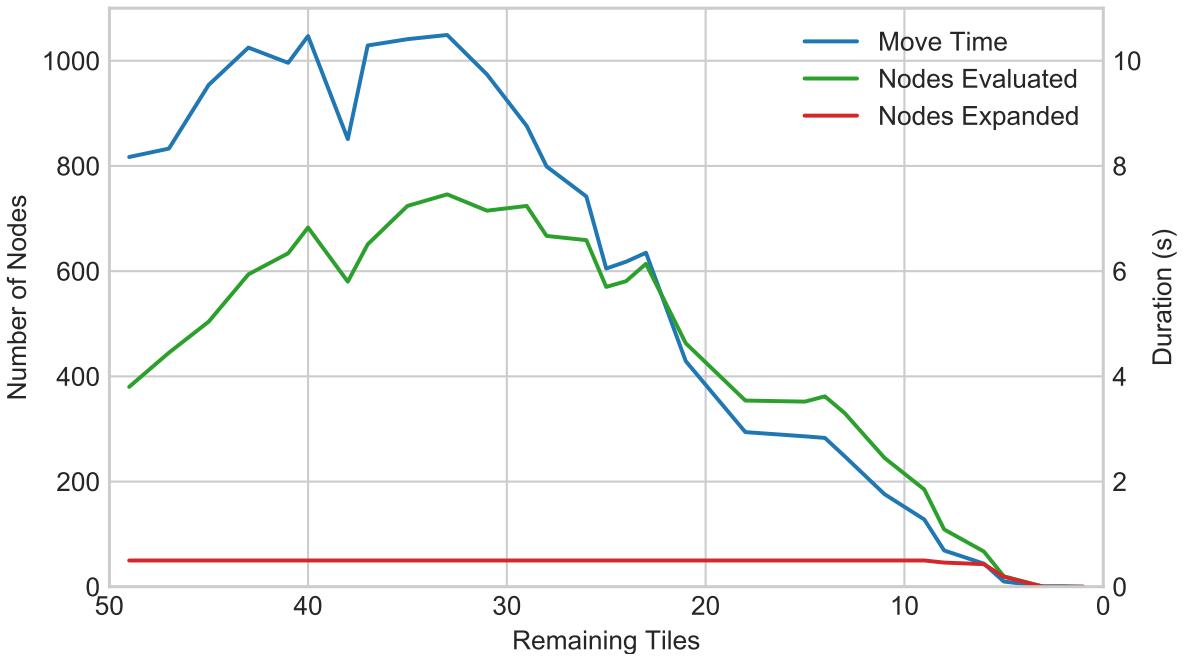

Figure 3.5: Move times etc. of `EtaZero-50` with the value-win network for one game of self-play. For a policy-value network, *Nodes Evaluated* would instead equal *Nodes Expanded*, capped at 50. Given the strong correlation between move times and nodes evaluated, we can see how policy-value is much more time-efficient.

This downside provides motivation for an alternative. During the MCTS explained in section 2.3, instead of evaluating state s and receiving a policy vector, we could evaluate each child of s, providing us with a *value* and a *win probability* for each one. We can then use the *win probabilities* from the children to form the policy vector, which of course should be normalised. If we train the *win probability* to be the value Q stored in the edge from s to the child, then in theory, the vector of *win probabilities* will be a useful prior probability for guiding the MCTS.

Whilst this method worked technically, it involved performing far more evaluations than the policy-value network during tree search. Figure 3.5 demonstrates the unnecessary computation of evaluating the children of the expanded nodes rather than just the expanded nodes (as would be the case for policy-value networks)[3]. Evaluating nodes is the main bottleneck[4], as shown by the correlation between move times and number of nodes evaluated. Given the computational costs, it was not considered worthwhile to proceed with the value-win network.

[3]See figure 4.3 to see the improved times of policy-value networks.

[4]Primarily due to the high cost of converting a state to its graph representation. Graphs of early-game states are computationally expensive due to the large number of relations between nodes.

3.3.3 2-Dimensional Convolutional Network

The beginning of section 3.2 explains how a 2D matrix is not the most appropriate representation of the game. Moreover, encoding each possible move in the output vector for the *policy* component is infeasible for Sevn. For base number 7, there are $7 \times 7 = 49$ ways to choose one tile. There are $\binom{49}{n}$ ways to choose n tiles. We need to encode all possible moves, hence the size of the output vector required is $\sum_{n=1}^{7} \binom{49}{n} \approx 10^8$. This would result in an extremely large number of parameters in the network; the network would probably be too big to train effectively with the resources available.

For these reasons, a 2D convolutional network was not chosen for implementation.

3.3.4 Relational Graph Convolutional Network (RGCN)

This implementation of an RGCN is based off a model theorised in 2017 [22] which considers how different nodes will have different relations between them. As such, for each layer in the network, there is a set of weights and biases for each relation type.

RGCN Description

The input of the network is a graph i.e. collection of edges and nodes. Each node contains a feature vector. Each edge is associated with a relation type and is directed. The formation of the graph and the 7 relations are described in section 3.2. Undirected relations in that section are implemented as two directed edges.

The network is composed of different layers, each of which has its own weights and biases. For a layer with input size n and output size m, the number of weights is $(r+1) \times n \times m$ where r is the number of relations (7 in our case). We add 1 to r to create a set of weights used for the identity relation.

Per layer, the data in the graph is transformed in 3 stages. The first is message passing. Let $\mathbf{w}$ be the 3D weight matrix for the current layer. For each edge $u \to v$ of relation i, we multiply the feature vector of u by $\mathbf{w}_i$ and pass the result to v.

The second stage is message aggregation. Each node will have received a number of messages (including the identity message from itself). Any aggregation function which converts a vector to a scalar can be used.

In the third stage, the result is applied to an activation function. The result becomes the new feature vector for the node.

After the transformation from all layers is complete, each node has a feature vector of size 2. To compute the *value*, the mean is taken of the first value in each vector. The *policy* vector is made up from the second value of the vector of each *action* node.

RGCN Hyperparameter Choice

With the core aim of the project being to investigate the feasibility of training AlphaZero rather than perfecting a trained instance of the prediction network, the exact choice of

hyperparameters is reserved for the evaluation chapter (table 4.1). The hyperparameters described here are related to design decisions of the network given the specific context of this project.

The number of layers in the network affect how many hops information can propagate through the graph before the output is reached. It is important for all starting information to be able to reach every node in the graph. For this, 7 layers is sufficient. The dimensions were `[3,64,64,32,32,16,8,2]`.

The choice for message aggregation was the max function. As demonstrated by the minimax algorithm [19], the optimal AI makes use of max aggregators rather than mean aggregators. That's because we are not interested in the losing moves, but only the best move. In the same way, when evaluating a position, we should value it the same as that of the best move, not the average evaluation of all moves. For this reason, the max aggregator was selected.

3.3.5 Training

The neural network's parameters are initialised via Xavier uniform initialisation [14] to θ_0. Xavier initialisation ensures the parameters aren't too big, which is important when the max aggregator function is used. The network is trained using the following loop:

1. The current network at iteration t with parameters θ_t is used in `EtaZero` to self-play N games. Each game played results in a set of training data examples: For each state s encountered in the game played,
   - Training example input: s.
   - *Policy* label: the vector of visits to each child state from s normalised.
   - *Value* label: 1 if the current player at state s won else -1.
2. The network is trained on the data just generated to create parameters θ_{t+1} which becomes the current network.

This approach was chosen over an approach which continues with a network instance only if it performs better than the current network, which is often used for ML projects such as genetic algorithms [12]. This is because the method of evaluation (described in section 3.5) is not absolute, but only an estimate of performance. There is a chance that an agent over-performs or under-performs during evaluation, each of which are equally likely. If we ignore instances which perform worse than the current network, then we may only continue when over-performance by chance occurs yet we ignore the under-performing cases, in which case our evaluation may show spurious improvement.

3.4 Agent Implementations

All agents described in this section inherit from an `Agent` class and are thus capable of participating in the game implementation described in section 3.6 against any other agent.

3.4.1 EtaZero

EtaZero[4] was implemented using the AlphaZero algorithm as described in section 2.3. Provided to the object upon initialisation are `network` (the function approximator $f_\theta(s)$), `samples_per_move` (the number of MCTS simulations performed per move), and a boolean flag signalling whether the instance is being used competitively or for training.

The implementation makes use of classes `StateNode` and `Action` to construct the tree of visited game states. `StateNode` has a method `expand()` which evaluates the current state using `network` and initialises its children nodes. `Action` stores P, N and Q required for move selection and has a method `update()` which updates N and Q during backtracking. Using these, the `EtaZero` class performs recursion on the game state tree, `expand`-ing and `update`-ing according to the AlphaZero algorithm.

To aid with code correctness and to help understand how EtaZero makes decisions, a class `EtaVisualiser` was developed. This class provides a user interface capable of exploring the EtaZero's constructed game state tree and the stored values after a game has been played.

The interface uses the `Renderer` class to display the state of the current node.

This interface was extremely useful for locating bugs in the code (specifically, identifying a missing minus sign) and understanding how the aforementioned balance of exploration and exploitation works numerically.

3.4.2 UCT

The `UCTAgent` class was implemented as described in section 2.2. Each `UCTAgent` instance is initialised with a parameter `samples_per_move` which determines how many MCTS playouts are performed before each move is chosen.

3.4.3 Other Agents

`RandomAgent` selects moves randomly according to a uniform distribution.

The `Human` agent is required to communicate with the user interface described in section 3.6. A user selects tiles to take using the mouse and presses a key to enter the move. The `Human` class takes a `UserInput` object which contains a `threading.Event` object. Because each agent runs on its own thread, separate from the user interface thread, this `Event` object is required. This object is signalled by the user interface thread once the space bar is pressed, after which the `UserInput` object contains the user-selected move which is safe to access. The agent thread blocks on this signal.

`RawNetwork` makes choices purely based on the prior probabilities given by a *policy-value* network, which is provided upon initialisation. The action selected is the one with the highest probability in the policy vector.

[4]Named after the seventh letter of the Greek alphabet η.

3.5 Arena

The arena was developed to battle agents against each other and calculate a quantitative measure of performance from the results. This section firstly describes the rating system used before describing how the arena was implemented.

3.5.1 Elo Rating

An Elo rating [8] provides a quantitative estimate of the strength of agents relative to one another. Such ratings allow us to derive an expected result of a game between two agents which haven't played each other.

The purpose of the arena is to pit agents against each other to calculate their Elo ratings. The Elo rating system is often used for zero-sum games such as chess and accounts for players whose strength varies over time.

Each agent has an Elo rating associated with them. Let two agents A and B have Elo ratings R_A and R_B respectively. The expected score (win ratio) of A in a game played by A and B is derived in equation (1):

$$E_A = \frac{1}{1 + 10^{\frac{R_B - R_A}{400}}} \tag{1}$$

E_A has range $(0, 1)$ where 1 corresponds to an expected win for A. Given the actual result S_A, the new rating for A is assigned as in equation (2):

$$R'_A = R_A + K(S_A - E_A) \tag{2}$$

R'_B is calculated likewise, bearing in mind that $S_B = 1 - S_A$. K is the development coefficient which determines how much the scores rely on recent performance vs overall performance. Equation (2) is the standard way to update a player's Elo rating. However, this is not the method used to update ratings in this project.

Interpretation of the Elo Rating System for this Project

This implementation makes an additional assumption: *agents' strength do not vary over time*. This assumption is used to reduce the number of games required to gain confidence in the results.

By assuming the performance of agents doesn't vary over time, we assume R_A and R_B are constant and hence E_A is constant. Equation (1) is rearranged to equation (3):

$$R_A = R_B - 400 \log_{10}\left(\frac{1}{E_A} - 1\right) \tag{3}$$

This is the formula used in the implementation. In this case, E_A is the observed win ratio of games between A and B. If this formula is used to calculate the rating of A, then A's

rating is dependant upon B's and this formula must not be used to calculate B's from A's. In fact, the dependency graph of all agents must have no cycles.

It is worth noting that the Elo rating system has a major pitfall – it assumes transitivity. Given data which says A is very likely to beat B and B is very likely to beat C, the rating system will tell you that A is extremely likely to beat C. Clearly this is incorrect for 3 agents whose strategies are rock, paper and scissors. This pitfall is inevitable for any rating system which attempts to rank agents on a linear scale. Bearing this in mind, whilst Elo ratings are very useful and informative, it is important to analyse it in conjunction with the win ratios and number of games played.

3.5.2 Arena Implementation

Our assumption that agents do not vary in performance over time requires us to treat each instance of EtaZero during training as a different agent. Therefore, we require an identification system – each agent instance must have an `elo_id` which distinguishes any agents which might have different strengths. Instances of the neural network during training are given IDs of the form:

```
<class name>-<iteration number>-<timestamp>
```

E.g. `PoicyValRGCN-7-2021-01-26-07-00-11`

These network instances are saved to disk. An `EtaZero` instance requires a network instance and an integer representing the number of MCTS samples to make per move. Of course that value affects the strength of the agent so must be included in the `elo_id`.

```
EtaZero-<samples>-<network id>
```

The `elo_id` of UCT agents are of the form `uct-<samples>`.

Arena battling can take up substantial time when using agents which take up to a minute to decide one move. For this reason I chose not to run the arena evaluation sequentially with the training loop described in section 3.3.5, but in parallel. The requirements of the arena are:

1. It can battle agents and record their history: number of games played and number of victories for each opponent.
2. It can calculate Elo ratings from the history.
3. When the Elo rating of an agent A changes, the rating of all agents on the subtree (where A is the root) of the dependency graph are updated.
4. It is automated – as the training loop proceeds, the arena detects new agents and decides which other agents to battle it against.
5. It decides to battle agents such that the dependency graph is acyclical.

Requirement 1 is satisfied by recording the history in a `json` file. Given that the game initialisation is random, it is possible that the initial game state favours either player one

or player two[5]. For that reason, during battling, games are played in pairs with the same state being used for both games in each pair but the first player is swapped.

It is mentioned above how an Elo rating is calculated against one opponent, but each agent may have played several opponents. This will lead to multiple ratings being calculated. To satisfy requirement 2, the final rating assigned is the mean of the ratings calculated from each opponent, weighted by the number of games played.

The algorithm to calculate ratings is given below. In order to satisfy requirement 3, it is run every time the history changes.

```
def calculate_ratings(history):
    ratings = {}

    @cache  # memoizes function calls
    def get_rating(elo_id):
        total_games = 0
        sum_elo = 0
        for enemy_id, wins, games in history.get(elo_id):
            elo = get_rating(enemy_id) - 400*math.log10(games/wins - 1)

            total_games += games
            sum_elo += elo*games

        ratings[elo_id] = sum_elo/total_games
        return ratings[elo_id]

    map(get_rating, all_elo_ids)

    return ratings
```

Requirement 4 requires performing tasks like *every `EtaZero` should battle the previous instance and the instance 10 iterations before it* or *every `EtaZero` should battle its closest `UCTAgent` by rating*. To achieve this, the `Arena` needs to abstract the notion of individual agents to all agents of a type e.g. a *series* of `EtaZero` agents. So, for each type of agent, I defined a `Series` class. `EtaZero.Series` contains a list of agent instances with increasing iteration number but fixed number of samples. `UCTAgent.Series` contains a list of agent instances with increasing number of samples.

The arena is implemented as a task scheduler. Firstly, tasks are added using `arena.add_task()` which specifies two series to battle each other and a shift parameter. Then, `arena.start()` is called which forever checks for new agents and performs the tasks specified. If we want each new `EtaZero` instance to battle against the previous one, we call `arena.add_task(EtaZero.Series(), EtaZero.Series(), 1)`.

Battling the `EtaZero` series with the `UCTAgent` series is a special case. It battles each `EtaZero` against a `UCTAgent` around the same rating and against one with a lower rating. Given the current rating R_η of a new `EtaZero` instance, the arena selects the closest

[5]The extent of which decreases as the base number increases

`UCTAgent` by rating to R_η. Secondly, the arena selects the closest `UCTAgent` (which wasn't selected first) to $R_\eta - 400$. A gap of 400 Elo corresponds to a $\frac{10}{11}$ estimated win ratio.

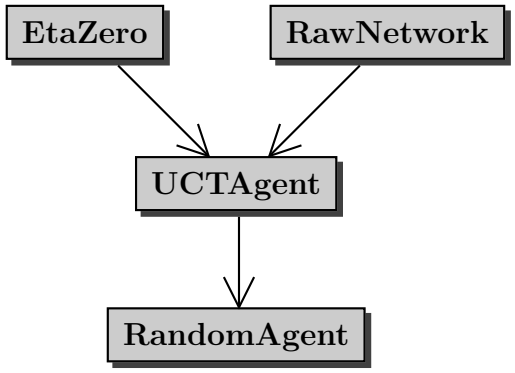

Figure 3.6: Series dependency graph. $A \to B$ means the Elo ratings of series A depend on those of series B. To ensure that Elo ratings stabilise, there must not be any cycles in the dependency graph. All games played in evaluation adhered to this dependency graph.

To satisfy requirement 5 and avoid dependency cycles, I only added tasks between series P and Q if there exists a path from P to Q in the dependency graph in figure 3.6. When $P = Q$, only positive shift is allowed i.e. an agent can only battle agents earlier in the series list. The rating system is calibrated so that `RandomAgent` has rating 500.

If `EtaZero` instances are only compared to previous instances, the Elo rating is likely to go wild since an inaccuracy in one rating is propagated directly to every subsequent rating. As such, it's important to compare to a variety of other agents to minimise the effects of inaccuracies. Each `EtaZero` agent at iteration i was battled against `EtaZero` agents at iteration $i-1$ and $i-10$ and two `UCTAgent`s as described above. Battles against `EtaZero` agents consisted of 40 games each and those against `UCTAgent`s were 40 games each, meaning 160 games of evaluation per new `EtaZero` instance. Section 4.1.1 of the evaluation chapter gives confidence intervals for this method.

3.6 Game Implementation

Implementing the game of Sevn was crucial for the project as an implementation is not available open-source. This involved creating the game state and an interface for agents to traverse through states during a game.

3.6.1 Game Class

The `Game` class is an interface connecting agents with the game logic and is responsible for instantiating new states as instructed by agents' moves. Most importantly, it has methods `make_move(move)`, `undo_move()` and `get_moves()`; the latter returns a collection of all possible moves from the current state.

3.6.2 State Class

The `State` class is an immutable representation of the game state. It is a composition of a `Board` and a `Score`.

A `Board` contains a 2D array of all remaining tiles on the board. This class is responsible for answering tile and colour queries, including providing all the takeable positions. The array is initialised randomly, ensuring there are the correct number of tiles of each colour.

For base number n, a `Score` contains a list of n integers `score`. Each integer is in the range $[-n, n]$. Player 1 taking a tile of colour i increments the ith integer and player 2 likewise decrements it. The array is initialised to zeros and any integer reaching $\pm n$ indicates a player has won by taking all tiles of a colour. If the board is empty and $\nexists x \in \texttt{score}.|x| = n$, then player 1 wins if more than half the values in score are positive, else player 2 wins.

The `State` class contains a method to convert to graph format and methods to convert to and from the string representation of the state. The string format is useful as it's a portable encoding of a state and is the format used in the training data saved to disk. The string representation `"1/aaa/aab.cba.cbc"` would result in the state seen in figure 3.7.

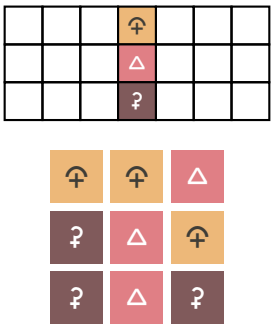

Figure 3.7: The state represented by `1/aaa/aab.cba.cbc`.

The grid above displays the score, with one counter for each colour. The counters move horizontally left when player 1 takes a tile of the corresponding colour and right when player 2 does likewise. The string representation has three segments:

- The `1` means player 1 is the next player to move.
- The second segment encodes the score. There is one character per colour. `a` means the score is 0, `b` means 1 etc. Negative scores are represented with a minus sign in front.
- The final segment encodes the tile positions. Rows are encoded left to right and are separated by full stops. The letters correspond alphabetically to the indexes of the colours in the score array. Numbers in this segment count consecutive empty cells.

As a second example, `"2/-dcc-ba/5.1dbbc.deeaa.eecb1.1ce2"` results the state shown in figure 3.8.

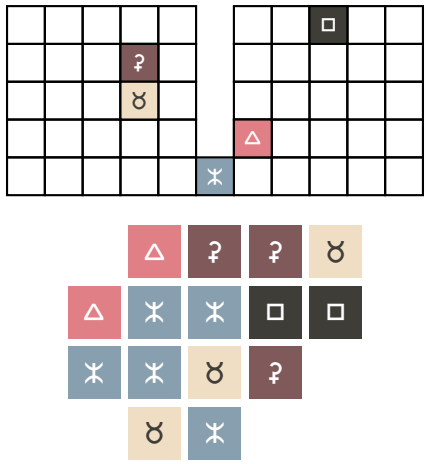

Figure 3.8: The state represented by `2/-dcc-ba/5.1dbbc.deeaa.eecb1.1ce2`.

3.6.3 User Interface

A user interface in this project is useful for visualising games, understanding the agents' strategies and allowing humans to play the game against any agent. The code to render the tile board and score grid are kept in a `Renderer` class. The UI is shown in figure 3.9. In order to ensure good visibility, the choice of colours and style was largely copied from the iOS App which is also shown in figure 3.9.

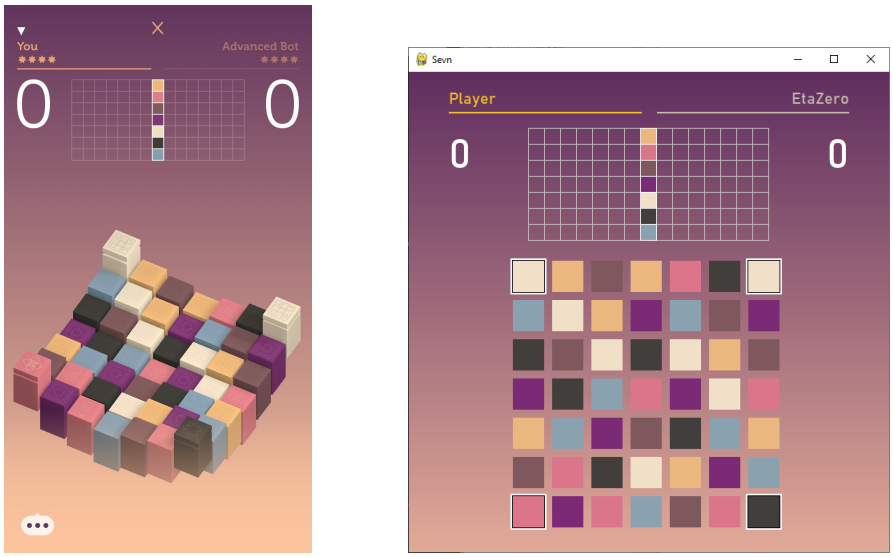

Figure 3.9: Screenshots of the app's (left) and the project's (right) user interface.

The interface provides the user with additional information, such as the progress of an agent choosing their move (the yellow bar under an agent's name is a progress bar) and an agent's confidence that they will win – these are optionally provided by agents. The string representation of every game state and the time taken for each agent to move is displayed in STDOUT.

3.7 iOS Screen Parsing

In order to evaluate EtaZero against the AI from the official Sevn app, the initial game state needs to be acquired. Given that the app gives you 30 seconds to make a move, it is not feasible to do this by hand. This section describes how I automated this process.

I screen-shared my phone to my computer and used Python module `win32gui` to capture the correct window. I predefined 49 pixel coordinates, one for each tile. Figure 3.9 shows an example screen capture of a starting state. Notice how the app uses 3D effects with shading. We want to ignore the effects of shading. To do that, we can convert the RGB values taken from the pixel samples to HSV, where the V component corresponds to the brightness of the colour. By dividing the V component by 100, the shading is largely ignored.

The final step is to group the 49 tiles into 7 clusters. The clustering algorithm exploits the fact that we expect 7 tiles in each cluster. The colours presented in the score grid at the top are used as reference colours – this is vital since the colour palette of the app changes with the difficulty of the bot. A priority queue is used to sort and choose tiles by proximity to a reference colour in HSV space. Tiles are greedily assigned this way, removing reference colours when their group becomes full.

This method successfully adapts to different colour palettes almost always gives the correct result.

3.8 Summary

This chapter explained the design decisions behind the neural network architecture, the game state representation and evaluation tools as well as describing how each component operates and interacts. Having developed the system, the AI was trained as described in section 3.3.5 on Google Colab with the arena (section 3.5) running concurrently as new agent instances were produced. This gave the results which are analysed in the evaluation chapter.

Chapter 4

Evaluation

The work described in the implementation chapter produced multiple series of agent instances and the results of many games between them. With the aim of assessing the level of mastery achievable and the feasibility of training an AlphaZero AI on ordinary hardware, this chapter analyses these results.

The original success criteria specified primarily revolve around the performance of the AI and how it compares to various other agents. Comparing the performance is also useful for assessing the success of the project relative to the aimed contributions to the field mentioned in the introduction and section 1.1. To quantify the performance of the AI and other agents, the Elo rating system was used, which is explained in section 3.5. Once we make comparisons of agents on their Elo ratings, we will be in a position to assess the success criteria.

Throughout this chapter, 95% confidence intervals are given. These are calculated by the method in section 4.1.1.

4.1 Elo Rating of EtaZero

This section analyses the results from the arena implementation, described in section 3.5. The arena automatically battled agents against each other to estimate their Elo ratings which acts as a quantitative measure of performance.

4.1.1 Comparisons to UCTAgent

Rating Comparison

Figure 4.1 shows the Elo rating of EtaZero after each training iteration. Each `UCTAgent` instance battled 120 games against other `UCTAgent` instances with fewer samples. Each `EtaZero-50` instance was evaluated by battling 80 games against earlier `EtaZero-50` iterations and 80 games against `UCTAgent`s.

In analysing figure 4.1, it is important to understand the change in amount of training data over time and observing how it affects the level of convergence reached. Table 4.1

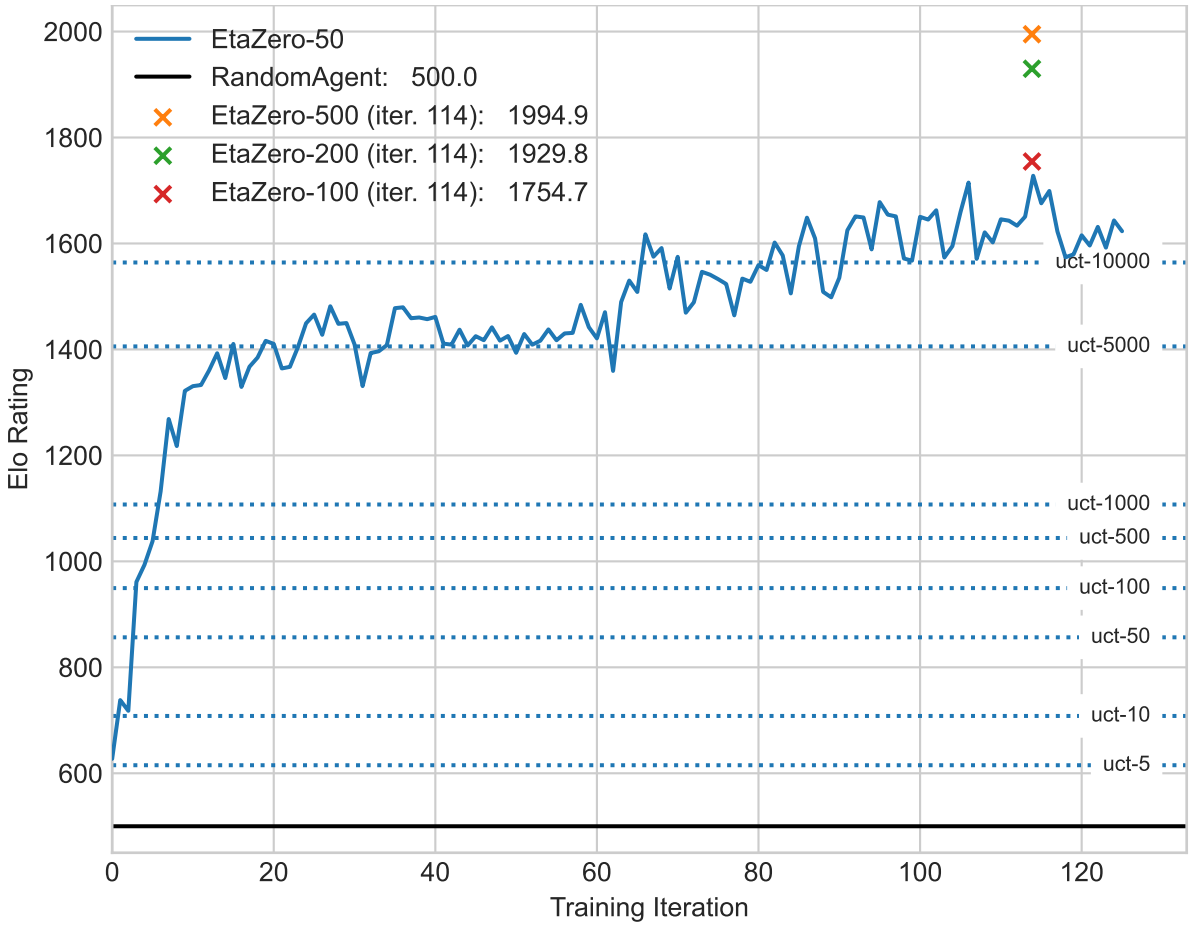

Figure 4.1: Elo rating of EtaZero over iteration number.

shows the number of games of self-play used to train each iteration and consequently the increasing number of training examples per iteration. Note that the number of games of self-play and the number of training examples are not directly proportional because different ratios of games of base 7 and 5 were used.

We can see the training has been very successful. `EtaZero-50` at iteration 114 had a rating 163.7 Elo above `UCTAgent-10000`, meaning a 72.0% expected win ratio, with a 95% confidence interval of 64.7 to 78.2%.

`EtaZero-500` (which was evaluated with 240 games) had greater performance with a 92.3% expected win ratio against `UCTAgent-10000`, with a 95% confidence interval of 89.0 to 94.6%.

Hyperparameter Choice

Given the primary goal of investigating the feasibility of training AlphaZero, this project aimed to find hyperparameters which resulted in success, rather than to find optimal ones. Table 4.1 shows the final choice of hyperparameters.

Table 4.1: Choice of hyperparameters for EtaZero's *policy-value* network.

Hyperparameter	Choice		
Network dimensions	`[3,64,64,32,32,16,8,2]`		
Epochs	10		
Learning rate	0.0003		
Batch size	64		
Loss	Mean-squared error		
Intermediary Activation	ReLU		
Final Activation	tanh		
Optimizer	SGD		
Samples per move	200		
	Iteration Range		
	1-59	60-79	80-
Self-play games per iteration	90	240	1000
Training examples per iteration	∼1400	∼4900	∼17000

Confidence Intervals of Ratings

How confident can we be in these Elo ratings? Figure 4.2 was plotted to investigate this. To statistically calculate a confidence interval, I created 10000 dummy agents which immediately win with a predefined win rate; the win rates were uniformly distributed in [0,1]. Given an observation of n games and x wins, I made each dummy agent play n games. To get a confidence of c%, I found a range of win ratios such that c% of dummy agents who won x games lie in that range. During evaluation, iteration 114 played 160 games and won 113 of them, hence figure 4.2 was plotted with this assumption. The results say we should be 95% confident in an interval of size 117.2 Elo (±58.6 Elo, that is). This method is used throughout the evaluation chapter to generate confidence intervals for win ratios, by using the maximum and minimum of the Elo range in the expected win ratio formula (equation (1) in section 3.5).

Confidence in these confidence intervals was instilled after I ran iteration 114's evaluation 8 times. The rating varied from 1611.9 to 1727.8 Elo, which gives a range of 115.9 – very close to our confidence interval size.

Our confidence interval is large enough that we can no longer be confident that iteration 114 is the best iteration that we've seen, but we can be very confident that the overall trend of figure 4.1 is correct.

Null Hypothesis Testing for Significance

Can we be confident that `EtaZero-50` (iteration 114) is significantly better than `uct-10000`? To investigate, we can perform a null hypothesis test. Of the 40 games `EtaZero-50` played against `uct-100000`, 32 were won. Let our null hypothesis be that this iteration of `EtaZero` is not significantly better than the `UCTAgent` i.e. the true win ratio is 0.5. For 40 games, the probability of observing at least 32 wins given a true win ratio p of 0.5 is given below.

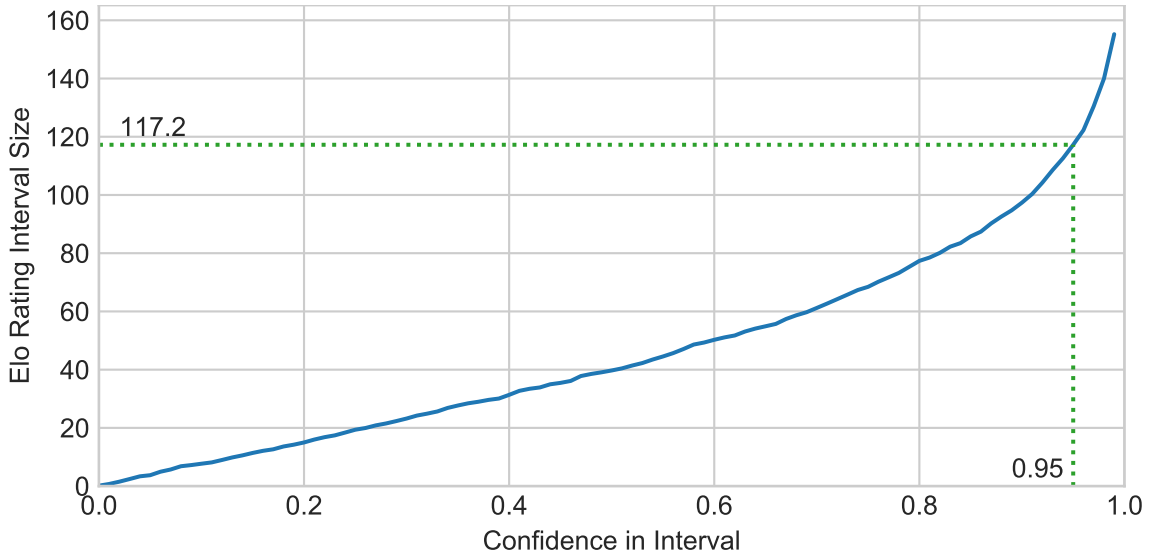

Figure 4.2: The size of confidence interval in Elo ratings over level of confidence of the interval. This plot assumes we observed 160 games and 113 wins.

$$
\begin{aligned}
P(wins >= 32 | p = 0.5) &= \sum_{w=32}^{40} \binom{40}{w} (½)^w (½)^{40-w} \\
&= \sum_{w=32}^{40} \binom{40}{w} (½)^{40} \\
&= 9.11 \times 10^{-5}
\end{aligned}
$$

This is small enough to pass any reasonable tail-test we throw at it. Therefore, we can be very confident that `EtaZero-50` (iteration 114) is statistically better than `uct-10000`

Resource-Based Comparison

Whilst figure 4.1 is very informative about the performance of agents evaluated, it is important to consider the resources that each agent consumes. Figure 4.3 shows the time it takes for various agents to make moves.

Figure 4.4 compares the ratings of `EtaZero` and `UCTAgent` on move times. We can see that `UCTAgent` with n samples roughly takes the same time to move as `EtaZero` with $\frac{n}{10}$ samples. From the examples on this graph, the average difference in Elo rating between `EtaZero-n/10` and `uct-n` is 638.25. **So, pitting agents equally on move times, we expect a win ratio of 97.5% of EtaZero over the UCT algorithm**. This is with a 95% confidence interval of 96.6 to 98.2%.

This proves that I have met the success criteria:

- EtaZero is able to play the game.

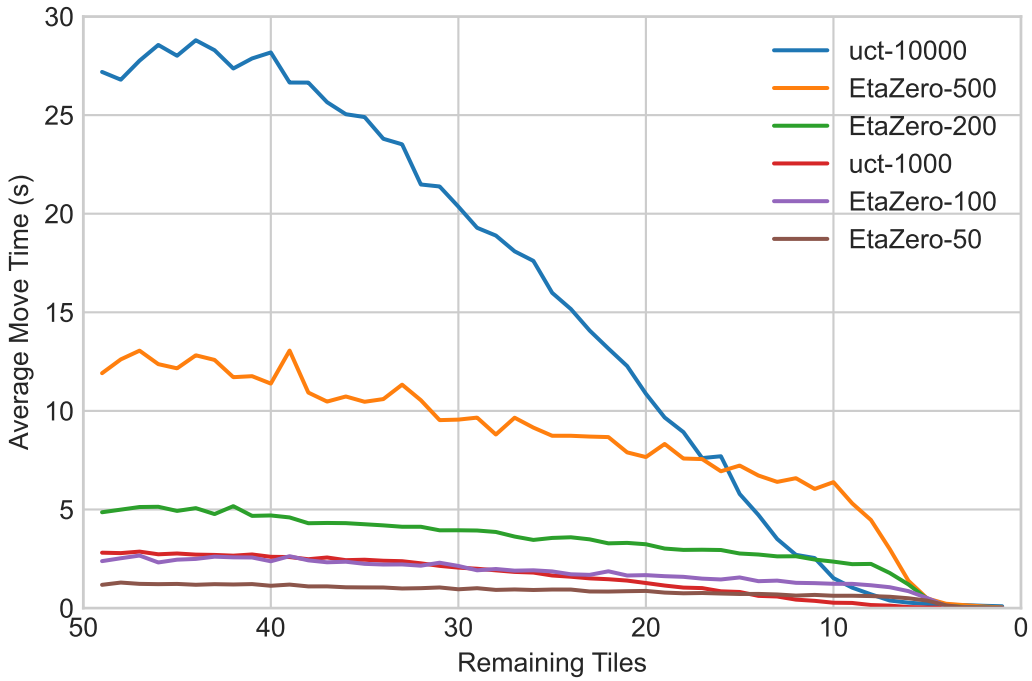

Figure 4.3: Average move times for various agents on Google Colab (Intel Xeon(R) CPU @ 2.30GHz). Each agent played 30 games to generate this data.

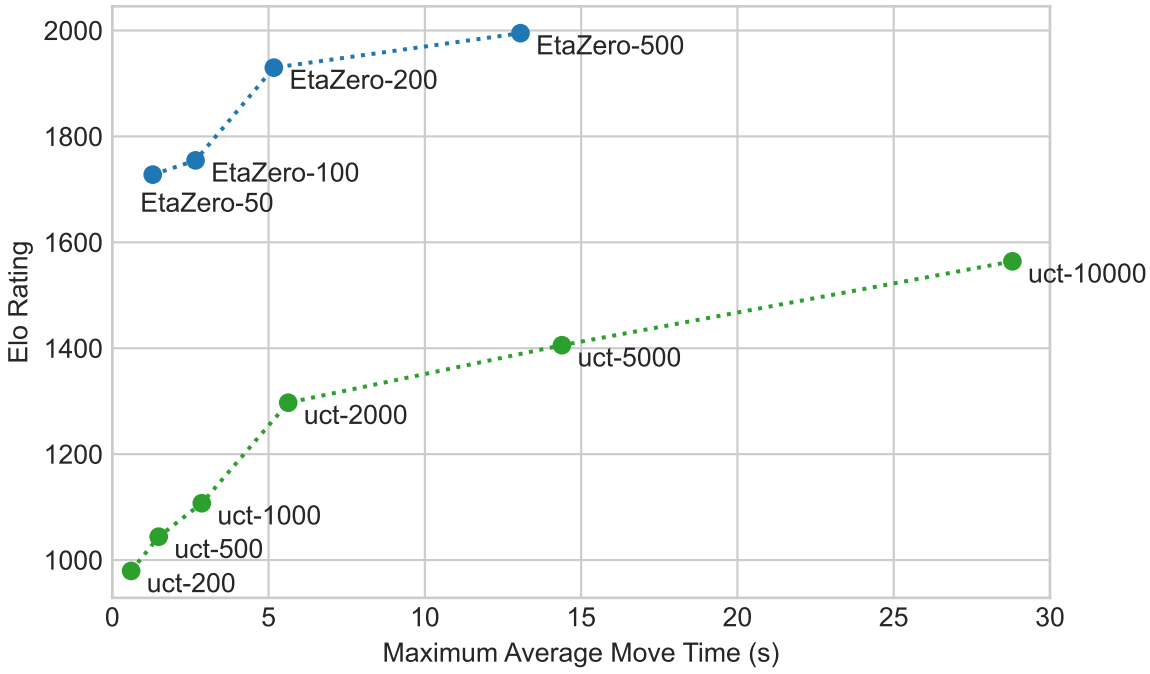

Figure 4.4: Elo rating of various agents over move times. `EtaZero` ratings are for iteration 114. The move times used were the maximum of the average move times in figure 4.3 for each agent.

- EtaZero is statistically better than RandomAgent.
- EtaZero is statistically better than a heuristical MCTS agent (UCTAgent).

UCT's Shortfall

As mentioned in section 1.1, the UCT algorithm performs poorly in the early game – it has no prior knowledge of the game and thus can only learn which moves are good through trial and error. At the early game, the remaining state-space size is too large to reliably confirm that certain moves are sensible and UCT spends its time exploring new states rather than exploiting known states. An example of misevaluation from `uct-10000` encountered during evaluation is shown in figure 4.5; of the four takeable pink (♙) tiles, UCT chose only one (circled in the figure). This is a bad choice because it is giving up free pink (♙) tiles which wouldn't have exposed any particularly valuable tiles for the opponent. A deep analysis with `EtaZero-50000` (iteration 114) confirms that this is a bad choice giving it an 8.7% win probability. EtaZero suggests taking all four takeable pink (♙) tiles for a win probability of 37.0%.

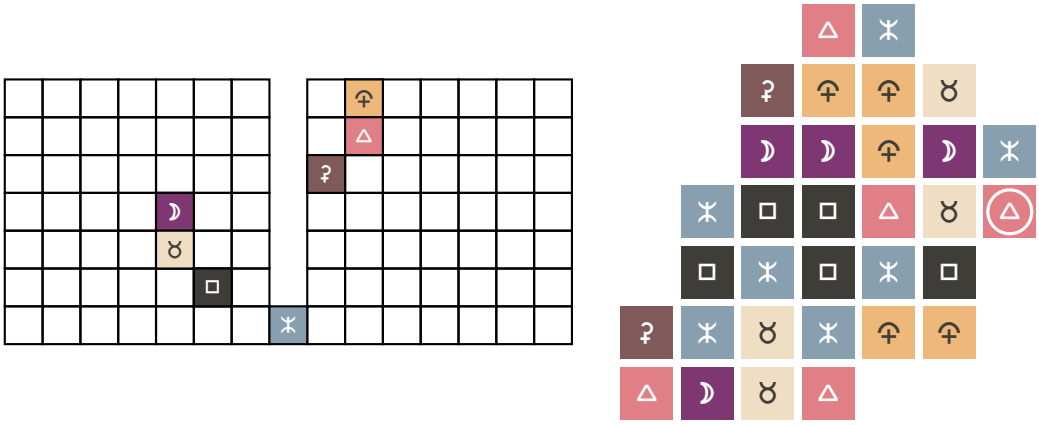

Figure 4.5: An example state where `uct-10000` misevaluated and took only the circled tile. The best move is to take all 4 exposed pink (♙) tiles.

For this reason, EtaZero is expected to outperform the UCT Algorithm. This section has shown evidence of this, meaning the results fit our expectations; therefore we can conclude that the implementation was a success and have shown evidence for graph neural networks being a feasible application of AlphaZero to ordinary hardware.

4.1.2 Comparison to iOS App Agent

The best agent in the app is called Prodigy Bot. Prodigy Bot was evaluated with 40 games against various instances of `EtaZero-50`. This gave an Elo rating of 1462.1 and thus a 6.3% expected win ratio against `EtaZero-200`[1]. The 95% confidence interval is 3.4 to 11.6%.

This proves that I have met the success criterion:

[1]This agent was chosen for fair comparison since they both make moves within 5s.

- The game state from my iPhone can be correctly captured, allowing my AI to play the AI from the app.

And that I have met the extension criterion:

- EtaZero is statistically better than the AI in the official app.

4.1.3 Comparison to Human Play

Unfortunately, it is not possible to quantitatively judge whether an AI for Sevn is better than all humans since the game is not widespread and there are no *world champions*. Instead I have been able to compare the app against myself (with a few hundred games of experience) and some beginner players[2]. To make a comparison, I calculated two Elo ratings: one for myself and one for all the beginner players collectively; of course, different beginner players will have different levels of performance, so this is just an indication about beginner players in general.

I played 30 games against various `EtaZero-50`s and 10 games against `uct-10000`. My Elo rating was 1476.6. I took up to 30s per move; so comparing to `EtaZero-500`, I have an expected win ratio of 4.8% with a 95% confidence interval of 2.6 to 8.7%.

Each beginner player had never played the game before but were taught and understood the rules. Each game played in the evaluation was one of the player's first ten games. After 52 games played, the Elo rating was 1183.1, meaning a 0.9% expected win ratio against `EtaZero-500` with a 95% confidence interval of 0.5 to 1.8%.

Overall, we have provided evidence that the following extension criterion has been met:

- EtaZero is statistically better than humans.

4.2 Network Forgetfulness and Amount of Training Data

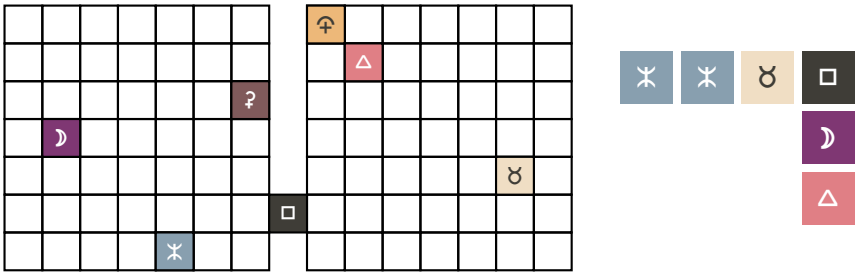

Figure 4.6: A simple example where the network suffered from forgetting. This state appeared in training, after which the network produced the correct answer, but then forgot it a few iterations later.

[2]Each of the seven beginner players were undergraduates at the University of Cambridge. To preserve the data privacy of the participants, no further information is given.

Why did increasing the amount of data per iteration improve performance? More data often results in better generalisation, but in this case, more data resulted in the network performing better on previously seen examples. The state shown in figure 4.6 was encountered during self-play while generating training data to create iteration 21. With player 1 to move, it is vital to pick the blue (⋇) tile. Taking the black (□) tile would expose the tile player 2 needs for victory (white (♉)). Taking the pink (△) tile would force player 2 to take the purple (◗) tile, after which player 1 needs both black (□) and white (♉) to win, but cannot take both against a sensible player.

Therefore a correct *policy* vector should score blue (⋇) as 1.0 and score both black (□) and pink (△) as 0.0. This is a winning position for player 1, so the correct *value* is 1.0. Figure 4.7 show the actual outputs of the network for each iteration. We can see that after the state was seen in training (iteration 21), the network gives a near-perfect output. This lasts for a few iterations before becoming a misevaluation again. However, after the amount of training data per iteration was increased to 1000 games per iteration (iteration 80 onwards), the network gives a correct evaluation much more consistently.

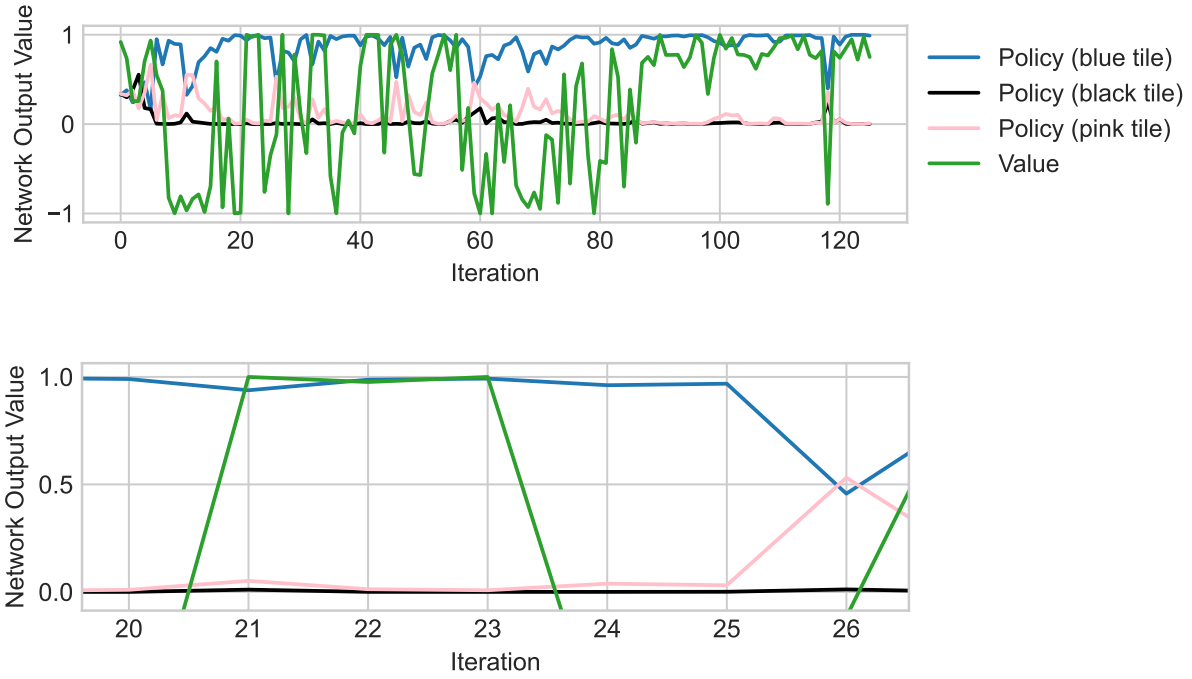

Figure 4.7: Network output values of each iteration for input state shown in fig. 4.6. The correct output is observed at iteration 21 when the state appeared in training, but catastrophic forgetting [18] shortly ensues. Iteration 21 and beyond is magnified beneath. Iteration 80 onwards, when the amount of training data was increased, is correct much more consistently.

4.3 Generalisation to a Higher Base Number

EtaZero was trained entirely on games of base number 5 and 7. The design of the neural network (section 3.3.4) and the game state representation (section 3.2) aimed to treat the

base number as an invariance. Therefore, we anticipate that EtaZero will be able to play games at base 9 and we hope that strategies learnt during training transfer to the higher base number.

To test this, I battled `EtaZero-50` (iteration 114) against `RandomAgent`, `uct-5000` and `uct-10000` with games of base 9; `EtaZero` won 40 out of 40, 37 out of 40 and 35 out of 40 games respectively. The latter is a win ratio of 87.5% which is an improvement over the win 80.0% win ratio observed against `uct-10000` for base 7.

We can be confident that the UCT algorithm generalises to the higher base number since it was not designed for/trained on any specific game and can play any classical game. Therefore we have evidence to say `EtaZero` can play higher base numbers and is still very strong. Hence the graph design of the state representation was successful and I have achieved the extension criterion:

- Preceding criteria are achieved for more complicated versions of the game (i.e. basing the game off of the number 9 or higher).

The preceding criteria being the following:

- EtaZero is able to play the game.
- EtaZero is statistically better than RandomAgent.
- EtaZero is statistically better than a heuristical MCTS agent (UCTAgent).

AlphaZero is general algorithm which can be applied to any two-player, perfect-information game with a perfect simulator and DeepMind demonstrated this by applying it to go, chess and shogi [26]. However, each new application requires training an instance from scratch; the instance trained for go would not be able to play go on a different board size. This project has shown that a good choice of network architecture can further the generalisation of a pre-trained instance of AlphaZero.

4.4 Risks

During evaluation it is possible that unconscious biases existed and contributed to the success demonstrated. Such risks can arise by subconsciously giving advantages to agents or evaluating to suggest an agent has better performance than it actually does. Considering this, some measures were taken to reduce the possibility of these risks.

Since games are initialised randomly, it is possible that a starting state favours player 1 or 2. The advantage a player has is greater the smaller the base number. For this reason, agents were only evaluated using games of base 7 or higher. To counter any advantage, games were played in pairs of the same starting state but alternating starting player (except when evaluating Prodigy Bot where the app selected the starting state).

When repeating a state, if a human was player 2 the first game, it is possible they remember the first move played by the opponent and repeat it in the second game when they are player 1. Therefore human players were always player 1 first, then player 2.

Section 3.3.5 explains how training was designed to avoid spurious increases of Elo rating.

I intentionally automated the choice of which agents to battle and ensured each instance plays the same number of games. This avoids any biases which would come with selecting myself which agents should battle each other.

When battling against humans, it is possible that biases were embedded in the play, particularly when the agent played against myself since I have motivation for the agent to perform well. I can claim that I played with no conscious biases but there isn't much that can be done about subconscious biases, so comparison against humans was not used as the primary method of evaluation.

4.5 Reproducibility

Use of *GitHub* and the dependency manager *Poetry* meant that any new user can clone the repository and run `poetry install` to create a virtual environment with all project dependencies installed.

Use of *Google Colab* meant that anyone can run the notebooks which clone and use the repository. At the time of writing, a demonstration notebook[3] which evaluates a new agent and plots Elo ratings can be run by anyone. My supervisor confirmed that the project is reproducible by running this notebook from his own machine.

4.6 Summary

EtaZero is indisputably the strongest agent tested in this evaluation. Comparing equally on move times, EtaZero outperformed every other agent and human. On top of completing every fundamental and preferable deliverable (from table 2.1), every success criterion and every extension criterion has been met. By achieving these criteria, we have demonstrated that the aimed contributions to the field have been successful. Therefore we can say with confidence that the project achieved what it set out to achieve and was a success.

[3]The URL for which has been omitted for the author's anonymity.

Chapter 5

Conclusion

Having analysed the results in the evaluation chapter, we are now in a position to make conclusions about the project and reflect on the work done.

5.1 Assessment of Contributions to the Field

The introduction of this dissertation listed three contributions to the field that this project aimed to make. This section assesses the success of these aims given the results observed.

Open-Source Implementation of AlphaZero to Sevn

The implementation was completed with every deliverable in the requirements analysis (section 2.5) achieved. The full implementation with all trained models were included in the repository uploaded publicly to GitHub and was shown to be portable and reproducible in section 4.5. Therefore this aim has been achieved.

Application of Graph Neural Networks to AlphaZero

Section 4.3 shows that the application of graph neural networks has not only been successful, but very beneficial for the generalisation of pre-trained AlphaZero instances. This is fundamentally due to the work done in defining a suitable graph representation for the game state, as described in 3.2. This approach required a more Sevn-specialised design (in the same way AlphaZero required game-specific adjustments to the neural network architecture [26]) but this can be seen as a trade-off for the gain in post-training generalisation.

Feasibility of AlphaZero with Limited Resources

This project aimed to investigate the feasibility of training an AlphaZero-style AI without the elaborate hardware used by DeepMind. For this, I intentionally chose free, publicly available hardware (Google Colab) for training. Figure 4.1 demonstrates what is possible with this hardware. Iterations 1-60 took roughly 1 hour to complete each. The graph

shows how the performance plateaued quickly at above 1400 Elo[1]; it is clear this level is achievable within 24 hours of training. The second and third section required longer training at roughly 4 hours and 12 hours per iteration respectively. This resulted in a total of about 620 training hours (less than 1 month). It is likely that the level of mastery acheived in this project could be achieved sooner if a longer training time was used from the beginning.

Therefore, we can conclude that for a game of comparable difficulty as Sevn, a strong level of play is definitely achievable in a short amount of training time. State of the art performance is approachable the more time is dedicated.

5.2 Lessons Learnt

This project tested both software engineering and computer science abilities to a great extent. As such, it has been an excellent learning experience.

It would be more informative to restart training from scratch with different amounts of data per iteration, rather than splitting the iterations into sections. With that approach, we could learn about the rate of convergence for different amounts and may discover that fewer training hours are required than what was used in this project.

By the end of the project, I realised the UCT and AlphaZero algorithms are more similar than I originally thought. This similarity is explained in sections 2.2 and 2.3. With this foresight, I would have implemented the two in a more similar style with more code-sharing between them. This would have resulted in further modularity and less code.

This project makes an abstraction for games, meaning other games could be added. In contrast, the OpenAI gym [10] makes an abstraction for environments. Since games are a subset of environments, taking this approach would have resulted in further modularity.

5.3 Future Work

This project utilised graph neural networks, but did not implement 2D convolutional networks since they are infeasible for this case, as explained in 3.3.3. It would be interesting to extend this project to another game to firstly, investigate the generality of the approach used, and secondly, directly compare the graph approach to the 2D matrix approach.

In 2020, DeepMind released a paper describing their work on MuZero [23]. MuZero takes a step in generalisation beyond AlphaZero; it does not require game rules nor a perfect simulator. This is very useful for more real-world scenarios where there isn't a perfect simulator. To achieve this, on top of the *prediction* network that AlphaZero uses, MuZero uses two more networks to predict future states, actions and rewards. It would be insightful to investigate how feasible it is to train MuZero without elaborate hardware, to assess whether using graph neural networks is practical and to compare the performance with an AlphaZero implementation.

[1]Note a higher rating is immediately achievable by increasing the number of samples.

Bibliography

[1] About - Stockfish - Open Source Chess Engine.

[2] AlphaGo | DeepMind.

[3] Elo Rating System - Chess Terms - Chess.com.

[4] Google DeepMind Challenge Match - Lee Sedol v AlphaGo - match report | British Go Association.

[5] Pytorch vs. Tensorflow: Deep Learning Frameworks 2021 | Built In.

[6] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[7] Ankan Banerjee. Fast perft on GPU (upto 20 Billion nps w/o hashing) - TalkChess.com, 2013.

[8] Sam Sloan Arpad E. Elo. *The Rating of Chess Players, Past and Present*. 2008.

[9] S. Barry Cooper and Jan Van Leeuwen. *Alan turing: His work and impact*. Elsevier Inc., 1 2013.

[10] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba Openai. OpenAI Gym. Technical report.

[11] Murray Campbell, A Joseph Hoane, and Feng-Hsiung Hsu. Deep Blue. Technical report, 2002.

[12] Jenna Carr. An Introduction to Genetic Algorithms. Technical report, 2014.

[13] Mogens Dalgaard, Felix Motzoi, Jens Jakob Sørensen, and Jacob Sherson. Global optimization of quantum dynamics with AlphaZero deep exploration. *npj Quantum Information*, 6(1):1–9, 12 2020.

[14] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. Technical report.

[15] R Herbrich, T Minka, and Thore Graepel. TrueSkill: A Bayesian skill rating system. *NIPS*, pages 569–576, 5 2006.

[16] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo Planning. Technical report.

[17] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. UCT: Improved Monte-Carlo Search. Technical report.

[18] Michael McCloskey and Neal J. Cohen. Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. *Psychology of Learning and Motivation - Advances in Research and Theory*, 24(C):109–165, 1 1989.

[19] J C C McKinsey. *Introduction to the Theory of Games*. Dover Books on Mathematics. Dover Publications, 2003.

[20] Sergey I Nikolenko and Alexander V Sirotkin. EXTENSIONS OF THE TRUESKILL TM RATING SYSTEM. Technical report.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury Google, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf Xamla, Edward Yang, Zach Devito, Martin Raison Nabla, Alykhan Tejani, Sasank Chilamkurthy, Qure Ai, Benoit Steiner, Lu Fang Facebook, Junjie Bai Facebook, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. Technical report, 2019.

[22] Michael Schlichtkrull, Thomas N Kipf, Vu Amsterdam pbloem, vunl Rianne van den Berg, Ivan Titov, and Max Welling. Modeling Relational Data with Graph Convolutional Networks Peter Bloem. Technical report, 2017.

[23] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. MuZero: Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. Technical report.

[24] Claude E Shannon. Chess State-Space Size: Programming a Computer for Playing Chess 1. Technical Report 314, 1950.

[25] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. AlphaGo: Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 1 2016.

[26] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. AlphaZero: Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. Technical report, 2017.

[27] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Van Den Driessche, Thore Graepel, and Demis Hassabis. AlphaGo Zero: Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 10 2017.

[28] John Tromp. The number of legal go positions. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10068 LNCS, pages 183–190. Springer Verlag, 2016.

[29] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, Zheng Zhang, Web Services, Aws Shanghai, and A I Lab. DEEP GRAPH LIBRARY: A GRAPH-CENTRIC, HIGHLY-PERFORMANT PACKAGE FOR GRAPH NEURAL NET-WORKS. Technical report.

[30] Robin J Wilson. *Introduction to Graph Theory*. John Wiley & Sons, Inc., USA, 1986.

Appendix A

The Game of Sevn

Sevn is a two-player, turn-based strategy game about taking tiles. It involves a 7x7 grid with each cell occupied by a coloured tile at the start of the game. There are seven colours and seven tiles of each colour randomly placed at the start. On one turn, the player may take any number of tiles greater than zero so long as the following conditions are satisfied:

- All chosen tiles are the same colour.
- Each tile chosen is a corner tile (has two or more adjacent edges exposed).

In figure A.1, all and only the purple (☽) tiles are takeable.

Any player who has taken all seven tiles of a particular colour wins. If all tiles are taken and no player has obtained seven tiles of one colour, the winner is the player who owns[1] the most colours. There is no draw-state.

To store the state of the game, we require the score for each colour. We maintain a vector `scores` of 7 integers in the range $[-7, 7]$. This vector is initialised to zeros and player 1 taking a tile of colour `i` increments `scores[i]` and player 2 likewise decrements it.

The strategy of the game arises from comparing the value of tiles you could take with the value of the opportunity for your opponent to take tiles which would be revealed.

[1] A player owns a colour if they have taken more tiles of that colour than their opponent.

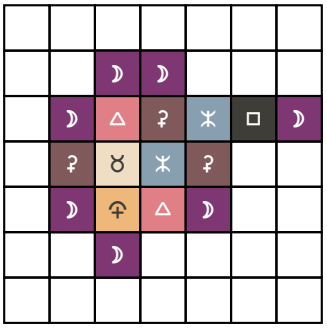

Figure A.1: An example state, with takeable tiles coloured purple (☽).

Appendix B

Project Proposal

CST Part II Project Proposal

Mastering Sevn using Reinforcement Learning

Introduction

Recent advances in artificial intelligence have demonstrated that mastery of many strategy-based board games far beyond human level is achievable. I propose to explore the level of mastery achievable of the game sevn. I propose creating a neural network-based artificial intelligence to play the game, described below.

The Game

Sevn is a two-player, turn-based strategy game. It involves a 7x7 grid with each cell occupied by a coloured tile at the start of the game. There are seven colours and seven tiles of each colour randomly placed at the start. It is a turn-based game about taking tiles. On one turn, the player may take any number of tiles so long as the following conditions are satisfied:

- All chosen tiles are the same colour.
- Each tile chosen is a corner tile (has two or more adjacent edges exposed).

Any player who has taken all seven tiles of a particular colour wins. If all tiles are taken and no player has obtained seven tiles of one colour, the winner is the player who owns[1] the most colours. There is no draw-state.

Sevn is a game designed by Dieter Stein and exists as a mobile app Sevn on the iOS App Store.

[1] A player owns a colour if they have taken more tiles of that colour than their opponent.

Appropriateness of a Deep Learning Solution

Deep learning is shown to be the best current approach for games with relevant similarities (e.g. chess, go). Several aspects of this game make it convenient to write AI for:

- Sevn is a classical game.[2]
- The number of moves in a game is capped at 49 ply[3], though is more typically around 30.
- The game state tree is directed and acyclic.
- There is no draw-state - one player must win.
- The complexity of the game can be scaled by basing it off of a number other than seven (though must be odd to retain the no-draw-state property).

The number of possible moves for a turn varies between 1 and 160 inclusive, though is typically less than 20 and is about 10 on average. We can then approximate a tree of game states with branching factor of 10 and depth of 30. The number of nodes (i.e. possible game states given a starting state) is then approximately 10^{30}. This is magnitudes less than chess (estimated between 10^{45} and 10^{46} unique states)[4], but ought to be sufficiently high for a deep learning solution to be more effective than a hard-coded strategy or heuristic-based approach.

Starting Point

I plan to implement this project from the ground, using no software I, or anyone else, has written before, except for the usage of libraries that will be determined in the preparatory stage of the project. This will likely consist of standard libraries for the chosen language(s) and a machine learning library.

I will research modern techniques for similar AI design such as AlphaZero to gain insight into the best approaches for my project.

Description of Substance and Structure

The Project

Given the relevant similarities to chess and go, an approach similar to DeepMind's AlphaZero can be applied.

An optimal AI is one which can correctly evaluate the value (i.e. the chance of winning) of a game state. Since traversing the entire game state tree is computationally unfeasible, I propose to create a solution with both deep learning and using exploration (semi-randomly

[2] A classical is a two player turn-based game with perfect information and a perfect simulation.
[3] One ply is one turn for one player.
[4] `https://math.stackexchange.com/questions/1406919/how-many-legal-states-of-chess-exists`

traversing) and exploitation (using previous exploration to exert confidence of a game state value) of the game state tree. If done correctly, such an approach will limit game state tree traversal to a computationally feasible size, focused on paths taken by competent players.

Monte Carlo tree search (MCTS) involves randomly selecting branches to traverse the game state tree. If a sufficient number of play-outs are completed, we can estimate the value of a game state to be the win percentage. However, this will give inaccurate results as it assumes the players will play randomly and will factor in many game paths which are only reachable by incompetent players. Thus we can aim to reduce the search space using exploitation; we greatly reduce the probability of choosing moves which MCTS has shown results in losses and hence exploit the moves which we are confident in (an approach known as upper confidence trees (UCT)).

This approach is not optimal, since two very similar game states are considered completely differently. A convolutional neural network can generate/contribute to a heuristic value to minimise this effect. The network will take the game state as input and will output an estimated value for the game state. I propose to research how such a network should be architectured for sevn, how it can be integrated with the above techniques and how I can use self-play to train it. I will evaluate the performance of the network alone and compare it to the performance of the network integrated into the tree search methods.

Since the game is rather niche, I will implement the game logic and visuals myself. To make it easy to test different agents and have different complexities of the game (e.g. basing the game off the number 9), I will write the code in a modular way with extensibility in mind.

Evaluation

We can assess how over-fitted the network is by reserving some generated data as test data and comparing the performance on the training and test data.

The only existing AI for the game that I am aware of is in the Sevn iOS app. To evaluate my AI against this AI, I will act as the middle-man between a game on my iPhone and a game on my laptop where my AI is running. However, due to the random starting state and a time limit on moves in the app, the starting state will have to be automatically determined. This can be done by streaming my iPhone to my laptop, taking pixel samples at predefined coordinates (at least one for each tile), and grouping the samples into seven colour groups.

As well as the existing AI, my AI can play against:

- Simpler AI which I create, e.g. a random-choice bot, a simple heuristic-based AI or a purely MCTS-based AI.
- Previous versions of itself.
- Human participants.

The performance against each AI/person can be evaluated by calculating the win percentage, and seeing if it's significant from a null hypothesis.

Success Criteria

Criteria 3 to 7 can be quantitatively determined using null hypothesis testing.

1. The AI is able to play the game.
2. The game state from my iPhone can be correctly captured, allowing my AI to play the AI from the app.
3. The AI is statistically better than a random-choice AI.
4. The AI is statistically better than a simple heuristic-based AI.
5. The AI is statistically better than a purely MCTS-based AI.

The following criteria are considered extension criteria since the level of mastery achievable without using elaborate hardware (such as the tensor processing units used to train AlphaZero) is unknown.

6. The AI is statistically better than the AI in the official app.
7. The AI is statistically better than humans.
8. Preceding criteria is achieved for more complicated versions of the game (i.e. basing the game off of the number 9 or higher).

Timetable and Milestones

Sprint 0: *12th October – 25th October*

Deadlines

- Phase 1 Proposal Deadline – 12th October, 3 PM.
- Phase 2 Proposal deadline – 16th October, 12 noon.
- Proposal Deadline – 23th October, 12 noon.

Sprint 1: *26th October – 8th November*

- Research relevant areas and papers.
- Select appropriate tools/languages.
- Begin game logic/visuals and agent structure.

Sprint 2: *9th November – 22nd November*

- Finish game logic/visuals and agent structure.

Sprint 3: *23rd November – 6th December*

- Implement tree search algorithms (e.g. MCTS, UCT)

Sprint 4: *7th December – 20th December*

- Using an ML library, implement and train the neural network.

Sprint 5: *21st December – 3rd January*

- Take a break; reflect on progress so far.

Sprint 6: *4th January – 17th January*

- Create evaluation tools: iPhone screen parsing and random/heuristic bots.

Sprint 7: *18th January – 31st January*

- Finish evaluation tools.
- Progress report.

Sprint 8: *1st February – 14th February*

- Finish progress report.
- Begin work for next sprint.

Deadlines

- Progress Report Deadline – 5th February, 12 noon.
- Progress Report Presentations – 11th, 12th, 15th, 16th February, 2 PM.

Sprint 9: *15th February – 28th February*

- Using evaluation tools, research and iterate on network performance and quality of data generation.
- Aim to surpass random/heuristic bot level.

Sprint 10: *1st March – 14th March*

- Same as above.
- Explore possibility of surpassing app AI level/human level.

Sprint 11: *15th March – 28th March*

- Finalise evaluation by performing null hypothesis tests.

Sprint 12: *29th March – 11th April*

- Dissertation write-up.

Sprint 13: *12th April – 25th April*

- Dissertation write-up.

Sprint 14: *26th April – 9th May*

- Dissertation write-up.

Sprint 15: *10th May – 23nd May*

Deadlines

- Dissertation Deadline – 14th May, 12 noon.
- Source Code Deadline – 14th May, 5 PM.

Resource Declaration

- My personal laptop - for development and evaluation of human/iOS app agents (Dell XPS 16 9570, i7-8750H CPU @ 2.20GHz, 16GB RAM).
- Google Colab - for training (Intel Xeon(R) CPU @ 2.30GHz, 13GB RAM)
- GitHub - for remotely versioning and backing up the repository.
- My personal iPhone 8 - for accessing the Sevn iOS app.
- OneDrive - for backing up my repository.
- Overleaf - for backing up my write-up.